Agrégation d'informatique 2024 – Sujet 1

Corrigé

⋆⋆⋆

1 Partie 1

Question 1.1 On peut imaginer l'exécution suivante :

- le fil 1 fait un appel à `libre` et lit la valeur de `i` ;
- le fil 2 fait un appel à `libre` et lit la même valeur de `i` ;
- le fil 1 fait l'appel à `aller(i)` puis `injecter(f)` ;
- le fil 2 fait l'appel à `aller(i)` puis `injecter(f)`.

La valeur de `i` étant la même pour les deux fils, ils ont bien déposé la solution dans la même éprouvette.

Question 1.2 Cette solution ne respecte toujours pas l'exclusion mutuelle : l'exécution précédente est toujours possible, quitte à encadrer les appels à `libre` par un verrouillage et déverrouillage du mutex, car l'éprouvette `i` reste libre tant qu'un fil n'a pas fait l'appel à `injecter`.

Question 1.3 Les instructions après un appel à `return` sont ignorées. Ainsi, le mutex ne sera jamais déverrouillé, donc il y aura une situation de famine pour tous les fils suivants.

Pour la corriger, il suffit d'inverser les lignes 8 et 9.

Question 1.4 La principale contrainte est d'éviter que l'appel à `encoder` se fasse pendant que le mutex est verrouillé. Comme cet appel ne dépend pas de l'indice `i`, on peut se contenter de reprendre la solution (corrigée) de la question précédente, et d'insérer la ligne 4 du code de base avant le verrouillage du mutex (en remplaçant `injecter(f)` par `injecter(&code)`).

Question 1.7 La présence du mutex garantit l'exclusion mutuelle : un seul fil peut faire bouger le bras et injecter à la fois, et on garantit que c'est toujours dans une éprouvette libre. De plus, il n'y a pas d'interblocage, car le mutex (unique) est toujours libéré après un temps fini. La fonction répond aux spécifications.

Question 1.8 En pratique, cette solution ne fonctionne pas du tout : il y a beaucoup d'attente active, et il est possible qu'un fil acquière le mutex en boucle. Même dans un cas où le mutex est FIFO (ce qui n'est pas garanti en pratique), cela peut être très lent.

Question 1.9 L'énoncé donnant une fonction `trouver`, on va implémenter une solution qui l'utilise : on déclare des variables globales de la manière suivante, où `N` est le nombre d'éprouvettes.

```
#define N ...

int tab[N] = {0};

sem_t libre;
pthread_mutex_t mutex;
```

Dans le `main`, on initialise le sémaphore à `N` et le mutex :

```
    sem_init(&libre, 0, N);
    pthread_mutex_init(&mutex, NULL);
```

Dès lors, le principe est d'attendre le sémaphore pour savoir s'il y a une éprouvette libre. Cela évite l'attente active et garantit qu'on trouve une éprouvette libre lorsqu'on acquière le sémaphore. On peut alors, après avoir verrouillé le mutex, trouver l'éprouvette, l'atteindre et injecter.

```
int deposer(struct formule* f){
    sem_wait(&libre);
    pthread_mutex_lock(&mutex);
    int i = trouver(tab, N);
    tab[i] = 1;
    aller(i);
    injecter(f);
    pthread_mutex_unlock(&mutex);
    return i;
}
```

La fonction `recuperer` fait ses opérations sous protection du mutex, en pensant à libérer le sémaphore.

```
struct resultat* recuperer(int i){
    pthread_mutex_lock(&mutex);
    aller(i);
    struct resultat* res = analyser();
    tab[i] = 0;
    pthread_mutex_unlock(&mutex);
    sem_post(&libre);
    return res;
}
```

Question 1.11 Un seul mutex est nécessaire : on le verrouille avant et on le déverrouille après les opérations.

```
void deposer(struct formule* f){
    pthread_mutex_lock(&mutex);
    int x, y;
    libreXY(&x, &y);
    allerX(x);
    allerY(y);
    injecter(f);
    pthread_mutex_unlock(&mutex);
}
```

Question 1.12 Il suffit de lancer un fil d'exécution qui fait le déplacement sur y pendant que le programme fait le déplacement sur x. Pour ce faire, il faut modifier la fonction `allerX` pour qu'elle prenne en argument un pointeur et non un entier.

```
void allerX(void* px){
    int x = *(int*) px;
    // code original de la fonction allerX
}
```

Dès lors, on peut écrire :

```
void deposer(struct formule* f){
    pthread_mutex_lock(&mutex);
    int x, y;
    libreXY(&x, &y);
    pthread_t filX;
    pthread_create(&filX, NULL, allerX, &x);
    allerY(y);
    pthread_join(&filX, NULL);
    injecter(f);
    pthread_mutex_unlock(&mutex);
}
```

Question 1.13 On a l'exécution suivante :

- T_0 : début de l'analyse en 5 ;
- $T_0 + 6$: début du déplacement vers 3 ;
- $T_0 + 26$: début de l'analyse en 3 ;
- $T_0 + 27$: début du déplacement vers 7 ;
- $T_0 + 31$: début de l'analyse en 7 ;
- $T_0 + 35$: début du déplacement vers 9 ;
- $T_0 + 37$: début de l'analyse en 9 ;
- $T_0 + 38$: fin de l'analyse en 9.

Question 1.14 L'énoncé n'étant pas extrêmement clair sur ce que doit faire la fonction `analyser_en`, on va supposer qu'elle fait à la fois le déplacement en i et l'analyse qui suit.

On propose une solution qui utilise :

- un tableau `attente` de `N` booléens indiquant quel fil veut lancer une analyse sur l'éprouvette i ;
- un tableau `sems` de `N` sémaphores, initialisés à zéro, permettant de gérer la priorité lorsque plusieurs fils sont en attente ;
- un mutex `mutex` pour protéger le tableau de booléens.

Dès lors, la fonction `analyser_en`, consiste à :

- déterminer si le fil courant est le seul à vouloir lancer une analyse (auquel cas il pourra faire son analyse) ;
- s'il n'est pas seul, il attend que son sémaphore soit libéré ;
- il fait alors son analyse (non protégée par le mutex, puisque soit il est seul, soit les autres sont en attente) ;
- enfin, il parcourt à nouveau le tableau de booléens, et libère le prochain fil qui est en attente, s'il existe.

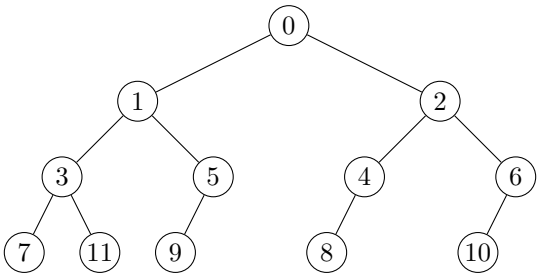
```
struct resultat* analyser_en(int i){
    bool libre = true;
    pthread_mutex_lock(&mutex);
    for (int j=0; j<N; j++){
        if (attente[j]){
            libre = false;
            break;
        }
    }
    attente[i] = true;
    pthread_mutex_unlock(&mutex);
    if (!libre) sem_wait(&sems[i]);

    aller(i);
    struct resultat* res = analyser();

    pthread_mutex_lock(&mutex);
    attente[i] = false;
    for (int j=i+1; j!=i; j = (j + 1) % N){
        if (attente[j]){
            sem_post(&sems[j]);
            break;
        }
    }
    pthread_mutex_unlock(&mutex);
    return res;
}
```

2 Partie 2

Question 2.1 Pour simplifier, on suppose que le tableau est constitué des entiers de 0 à 11. On obtient l'arbre :



Question 2.2 L'idée est d'aller à gauche ou à droite selon la parité de l'indice, et de recommencer, en modifiant l'indice pour qu'il corresponde au sous-tableau correspondant.

```
let rec get i t = match i, t with
    | 0, N(_, x, _) -> x
    | _, N(l, x, r) ->
        if i mod 2 = 1 then
            get ((i - 1) / 2) l
        else
            get ((i - 2) / 2) r
    | _ -> assert false;;
```

Question 2.3 On commence par remarquer que pour tout $n \in \mathbb{N}$, il y a une unique structure de tableau flexible (par récurrence simple), et que la hauteur est croissante avec la taille.

Montrons par induction que pour t un tableau flexible non vide, $|t| \geqslant 2^{h(t)}$:

- le résultat est vrai pour l'unique arbre de taille 1 (hauteur 0) et 2 (hauteur 1) ;
- supposons le résultat vrai pour ℓ et r deux tableaux flexibles non vides. Soit $t = N(\ell, x, r)$. un tableau flexible. On distingue :
  * si $|\ell| = |r| + 1$, alors $|t| = 2|\ell| \geqslant 2^{h(\ell)+1} = 2^{h(t)}$;
  * si $|\ell| = |r|$, alors $|t| = 2|\ell| + 1 \geqslant 2^{h(\ell)+1} + 1 = 2^{h(t)} + 1 \geqslant 2^{h(t)}$.

On conclut par induction.

Dès lors, on en déduit que $h(t) \leqslant \log_2(|t|)$, ce qui répond à la question.

Question 2.4 On remarque que l'argument de la fonction `liat` a une hauteur qui diminue d'au moins 1 à chaque itération. L'arbre étant de hauteur logarithmique en n, on en déduit une complexité totale en $\mathcal{O}(\log n)$.

Question 2.5 Par récurrence sur n :

- le cas de base `N(E, x, E)` doit bien renvoyer `E` ;
- sinon, on distingue :
  * si $n = 2k$ est pair, alors il faut éliminer l'élément d'indice $2k - 1$, qui est impair, et donc le dernier élément de ℓ, qui est de taille $k =$ `n / 2` ;
  * si $n = 2k + 1$ est impair, alors il faut éliminer l'élément d'indice $2k$, qui est pair, et donc le dernier élément de r, qui est de taille $k = \frac{n-1}{2} =$ `n / 2` (ici c'est une division entière).

Question 2.6 Le cas de base est facilement géré. Si on n'est pas sur un arbre vide, on rajoute l'élément x à gauche ou à droite, selon que l'arbre gauche soit plus grand que le droit ou non. On calcule bien les tailles pour les appels récursifs.

```
let rec snoc n t x = match t with
    | E -> N(E, x, E)
    | N(l, y, r) ->
        let nr = (n - 1) / 2 in
        let nl = n - 1 - nr in
        if nl = nr then
            N(snoc nl l x, y, r)
        else
            N(l, y, snoc nr r x)
```

Question 2.7 On commence par écrire une fonction qui calcule la racine :

```
let root = function
    | N(_, x, _) -> x
    | _ -> assert false
```

Dès lors, bien qu'astucieuse, la fonction `tail` est relativement simple à écrire :

```
let rec tail = function
    | N(E, _, E) -> E
    | N(l, _, r) -> N(r, root l, tail l)
    | _ -> assert false
```

Ici, on a la garantie que `l` est non vide lorsqu'on récupère sa racine. On constate qu'on a inversé les rôles de r et ℓ dans la reconstruction : si on note $\ell' =$ `tail`ℓ, alors sachant que $|r| \leqslant |\ell| \leqslant |r| + 1$, on a $|\ell'| = |\ell| - 1 \leqslant |r| \leqslant |\ell| = |\ell'| + 1$. L'inversion des arbres permet bien de garantir que l'arbre reconstruit est un arbre flexible. De plus, l'ordre des élément est bien respecté, car c'est bien la racine qui devient le

premier élément, et les éléments d'indices $2, 4, \ldots$ deviennent ceux d'indices $1, 3, \ldots$, et ceux d'indices $3, 5, \ldots$ deviennent ceux d'indices $2, 4, \ldots$.

La complexité est bien logarithmique, car l'appel récursif se fait sur un arbre de hauteur 1 de moins.

Question 2.8 La fonction `cons` utilise le même principe d'inversion des deux arbres, mais avec un ajout cette fois-ci :

```
let rec cons x = function
    | E -> N(E, x, E)
    | N(l, y, r) -> N(cons y r, x, l)
```

Question 2.9 On utilise l'indication en construisant la fonction auxiliaire demandée. Il suffit de faire le bon décalage des indices lors des appels récursifs. Dès lors, on fait un appel avec $m = 1$ et $k = 0$ pour avoir tous les indices $i = 1 \times i + 0$.

```
let of_array tab =
    let n = Array.length tab in
    let rec aux m k =
        if k >= n then E
        else N(aux (2 * m) (k + m), tab.(k), aux (2 * m) (k + 2 * m)) in
    aux 1 0
```

Question 2.10 Ici, une approche naïve pourrait consister à créer le tableau de taille `n / 2` et à insérer éventuellement un élément d'un côté avant de reconstruire le nouvel arbre. Malheureusement, une telle fonction aurait une complexité en $\mathcal{O}((\log n)^2)$. Pour améliorer la complexité, on écrit une fonction auxiliaire qui construit les arbres de taille n et $n + 1$ simultanément.

```
let make n x =
    let rec make2 = function
        | 0 -> E, N(E, x, E)
        | n ->
            let t1, t2 = make2 ((n - 1) / 2) in
            if n mod 2 = 0 then
                N(t2, x, t1), N(t2, x, t2)
            else
                N(t1, x, t1), N(t2, x, t1) in
    fst (make2 n)
```

La complexité temporelle est bien logarithmique en n, car la valeur de l'argument est au moins divisée par deux à chaque appel récursif. La complexité spatiale est également logarithmique en n, car elle correspond à la taille de la pile d'appels récursifs (et ne peut pas être supérieure à la complexité temporelle).

3 Partie 3

Question 3.1 Cela garantit qu'un même candidat ne peut pas s'inscrire à l'agrégation d'informatique dans plusieurs académies, ou avec chacune des deux options.

Question 3.2 On fait attention à l'ambiguïté sur les noms de champs.

```
SELECT nom, COUNT(*) FROM Académie AS ac
JOIN AgrégExtInfo AS ag ON ac.aid = ag.aid
GROUP BY ac.aid
```

Question 3.3 On joint plusieurs fois les mêmes tables pour faire la vérification. On s'assure que les identifiants d'académies sont bien distincts pour un même candidat.

```
SELECT c.nom, ac1.nom, ac2.nom FROM Candidat AS c
JOIN AgrégExtInfo AS ag ON c.cid = ag.cid
JOIN CapesNSI AS cap ON c.cid = cap.cid
JOIN Académie AS ac1 ON ac1.aid = ag1.aid
JOIN Académie AS ac2 ON ac2.aid = cap.aid
WHERE ac1.aid != ac2.aid
```

Question 3.4 On aurait pu faire des jointures, mais de simples tests d'appartenance simplifient la requête.

```
SELECT profession, COUNT(*) FROM Candidat
WHERE cid IN AgrégExtInfo
AND cid NOT IN CapesNSI
GROUP BY profession
```

Question 3.5 On fait une sous-requête pour calculer le nombre total de candidats inscrits à l'épreuve « Étude de cas informatique ». Pour calculer la proportion, on pense à convertir en flottant pour éviter la division entière (on pourrait utiliser `CAST`, mais ce n'est pas au programme).

```
SELECT COUNT(*) * 1.0 /
    (SELECT COUNT(*) FROM Candidat AS c
     JOIN AgrégExtInfo AS ag ON c.cid = ag.cid
     WHERE ep = 'A') FROM Candidat AS c
JOIN AgrégExtInfo AS ag ON c.cid = ag.cid
WHERE ep = 'A' AND genre = 'F'
```

Question 3.6 Attention, cela ne correspond pas nécessairement à la différence entre les noms des candidats ayant une salle isolée et les candidats ayant un tiers-temps (car il pourrait y avoir des homonymes). On pourrait éventuellement faire une différence avec les numéros de candidats, mais on choisit ici un test d'appartenance.

```
SELECT nom FROM Candidat AS c
JOIN ListeAmén AS la ON la.cid = c.cid
JOIN Aménagement AS a ON a.amid = la.amid
WHERE amén = 'Salle isolée'
AND (c.cid, (SELECT amid FROM Aménagement
             WHERE amén = 'Tiers-temps'))
    NOT IN ListeAmén
```

Question 3.7 On calcule les identifiants des académies qui ont des candidats en tiers-temps, puis on fait la différence entre tous les identifiants et cette table. On l'utilise alors pour récupérer les noms d'académies.

```
SELECT nom FROM Académie
WHERE aid IN
        (SELECT aid FROM Académie
         EXCEPT
         SELECT DISTINCT aid FROM AgrégExtInfo AS ag
         JOIN ListeAmén AS la ON la.cid = ag.cid
         JOIN Aménagement AS a ON a.amid = la.amid
         WHERE amén = 'Tiers-temps')
```

Question 3.8 On écrit une requête `R` qui calcule le nombre maximal d'aménagements "Salle isolée" pour l'agrégation, pour une même académie. Notons qu'on aurait pu utiliser `MAX` avec une sous-requête au lieu du

ORDER BY et LIMIT.

```
SELECT COUNT(*) FROM AgrégExtInfo AS ag
JOIN ListeAmén AS la ON la.cid = ag.cid
JOIN Aménagement AS a ON a.amid = la.amid
WHERE amén = 'Salle isolée'
GROUP BY aid
ORDER BY COUNT(*) DESC
LIMIT 1
```

S'il n'y avait pas eu de cas d'égalité, on aurait pu utiliser la requête précédente légèrement modifiée pour trouver l'académie correspondante. Comme l'énoncé suggère qu'il y a plusieurs académies correspondantes, on utilise la requête précédente pour les trouver :

```
SELECT nom FROM Académie AS ac
JOIN AgrégExtInfo AS ag ON ag.aid = ac.aid
JOIN ListeAmén AS la ON la.cid = ag.cid
JOIN Aménagement AS a ON a.amid = la.amid
WHERE amén = 'Salle isolée'
GROUP BY ac.aid
HAVING COUNT(*) = R
```

Question 3.9 On trie par somme de notes décroissante et on se limite aux 42 premiers résultats.

```
SELECT nom FROM Candidat AS c
JOIN AgrégExtInfo AS ag ON ag.cid = c.cid
ORDER BY note-ep1 + note-ep2 + note-ep3 DESC
LIMIT 42
```

Question 3.10 En utilisant une sous-requête donnant les identifiants des candidats ayant au moins deux aménagements, on peut calculer la moyenne de ces candidats.

```
SELECT AVG(note-ep1 + note-ep2 + note-ep3) FROM AgrégExtInfo
WHERE cid IN
    (SELECT cid FROM ListeAmén
     GROUP BY cid
     HAVING COUNT(*) > 1)
```

Question 3.11 On propose de garder les tables Aménagement, ListeAmén, Candidat et Académie telles quelles. On rajoute les tables suivantes :

- Concours décrit les différents concours et contient les champs :
  - concid (clé primaire) : identifiant unique du concours ;
  - nom : décrit le concours (exemple : 'Agrégation Externe Informatique')
- Inscription détermine quel candidat s'est inscrit à quel concours, et contient les champs :
  - concid (clé étrangère) : identifiant du concours ;
  - cid (clé étrangère) : identifiant du candidat ;
  - aid (clé étrangère) : identifiant de l'académie d'inscription ;
  - option : champ permettant d'indiquer l'option choisie, le cas échéant (pouvant être NULL)

  La clé primaire de cette table est le couple (concid, cid) : un candidat ne peut s'inscrire qu'une seule fois par concours.
- Notes permet de saisir les notes des candidats, par concours et contient les champs :
  - épreuve : nom de l'épreuve ;
  - concid (clé étrangère) : identifiant du concours ;

* cid (clé étrangère) : identifiant du candidat ;
* note : note obtenue à l'épreuve

La clé primaire de cette table est le triplet (épreuve, concid, cid).

Tous les champs sont de type varchar, sauf note qui est real.

Question 3.12 La requête demandée est alors relativement simple à obtenir.

```
SELECT COUNT(*) FROM Inscription
GROUP BY cid
ORDER BY COUNT(*) DESC
LIMIT 1
```

⋆ ⋆ ⋆