**Nom : _____ ( _____ /100)**

**Matricule : _____**

**Signature :_____**

# PCP – Contrôle final

Ceci constitue le contrôle final pour le cours ***IFT630 – Processus concurrents et parallélisme*** (PCP). Il est présumé, avant d'attaquer la présente, que vous avez pris une part active à l'intégration des concepts du cours dans vos travaux pratiques, et que vous avez fait les exercices qui, parmi ceux proposés, vous ont aidé à parfaire votre compréhension des éléments de matière qui vous semblaient un peu plus nébuleux.

Ce contrôle est récapitulatif, ce qui signifie qu'il couvre l'ensemble de la session (ce qui ne veut pas dire qu'il en couvre chaque parcelle, l'espace et le temps n'étant pas élastiques à ce point). En conséquence, gérez votre temps intelligemment.

## *Consignes*

Les consignes pour ce contrôle sont :

- le droit aux notes de cours est accordé;
- l'envoi de notes écrites, par télépathie, à l'aide de pigeons voyageurs, hiboux, outardes, patineurs de vitesse ou tout autre mode de communication est interdit;
- la calculatrice est bannie, de même que le téléphone cellulaire, l'ordinateur portatif, le *Walkie Talkie* et leurs dérivés;
- si vous bloquez sur une question, passez à la suivante. Il est plus sage de revenir plus tard sur ce qui vous pose problème, s'il vous reste du temps;
- les questions de ce contrôle n'ont pas toutes le même poids. Planifiez la manière dont vous investirez temps et efforts;
- une réponse précise, claire et véridique est préférée à une réponse imprécise, floue et fausse;
- la somme des points qu'il est possible d'amasser dans cet examen est supérieure à `100`, mais la note finale attribuée pour l'examen sera plafonnée à `100`;
- à moins d'indications contraires, vous pourrez répondre par du texte, du code, des schémas, etc. mais je ne pourrai corriger convenablement que si vos réponses sont claires. Il est donc à votre avantage de soigner le tout, et d'être aussi précis et limpide que possible;
- si la tendance se maintient, vous aurez une longue et heureuse carrière!
- répondez aux questions, simplement. Chaque réponse sera corrigée en fonction de l'énoncé de la question, alors assurez-vous de bien répondre à ce qui est demandé, pas à ce qui aurait peut-être pu l'être;
- pas de stress, la vie est belle. Sans blague. Et oui, je vous souhaite un bel automne!
- répondez directement sur le document, aux endroits prévus à cet effet;
- et surtout, prenez soin de vos animaux de compagnie!

***Que de plaisir en perspective!***

## Q00 Miaulements répartis (_____/20)

Le problème toujours croissant du nombre de chats errants et de leur reproduction « hors de contrôle » est devenu une question d'intérêt national depuis que le projet de loi 54 sur le bien-être animal a été adopté[1]. Alors que vous quittez le milieu universitaire pour celui du marché du travail, votre souci de contribuer au bien-être collectif vous amène à accepter une offre d'une entreprise nommée Minettes & matous, à la recherche d'un(e) spécialiste de programmation de systèmes parallèles et concurrents.

On vous informe tôt dans le processus que le fait que vous ayez réussi avec brio le cours `IFT630` a influencé l'intérêt que Minettes & matous a porté à votre candidature. Vous serez appelé(e) à participer à la conception et aux tests d'un nouveau système de micropuces intelligentes pour faciliter le suivi des chats errants. Là où les micropuces « traditionnelles » servent surtout à identifier un animal qui aurait été retrouvé dans la nature[2], celles de Minettes & matous seront géolocalisées, et permettront de localiser les chats dans la nature.

* * *

L'un des problèmes auxquels font face les organismes visant la stérilisation et la protection des animaux errants est leur repérage. Si tous les chats étaient géolocalisables, ce problème disparaîtrait pour l'essentiel, mais évidemment, Minettes & matous sait bien que cet idéal ne sera pas atteint, du moins pas à court terme.

Œuvrant de concert avec les villes et les organismes susmentionnés, Minettes & matous travaille à un modèle mathématique de distribution des félins dans une région, dans l'optique de mieux cibler les efforts des intervenant(e)s.

Étant donné que chaque ville est différente, avec ses propres quartiers, ses propres agglomérations, ses propres commerces, et ses propres points d'intérêts félins par le fait même, il importe d'adapter l'examen des dispositions géographiques des félins en tenant compte de ces caractéristiques locales. Par contre, une partie algorithmique plus générale peut être réutilisée d'une ville à l'autre, et c'est sur cette partie que porte votre contribution personnelle.

[1] http://www.lapresse.ca/actualites/politique/politique-quebecoise/201512/04/01-4927956-quebec-adopte-une-loi-pour-le-bien-etre-et-la-protection-des-animaux.php

[2] http://www.spca.com/?p=9871&lang=fr

La fonction `cibler_lieux()` ci-dessous prend en paramètre une séquence d'endroits (paramètre `lieux`), une séquence de félins (paramètre `matous`) et un prédicat générique à trois paramètres (paramètre `pred`). Ce prédicat sera ce qui variera selon les villes et tiendra compte des paramètres locaux. La fonction `cibler_lieux()` a pour rôle d'examiner chaque lieu `e`, d'appeler `pred()` avec e et la séquence de félins et de ne conserver que les lieux pour lesquels ce prédicat s'avère :

```
class Endroit { /* ... */ };
class Matou { /* ... */ };
namespace sequentiel {
   template <class Pred>
      vector<Endroit> cibler_lieux(
         const vector<Endroit> &lieux, const vector<Matou> &matous, Pred pred
      )
      {
         vector<Endroit> lieux_cibles;
         for (const auto &e : lieux) {
            if (pred(e, begin(matous), end(matous))
               lieux_cibles.emplace_back(e);
         }
         return lieux_cibles;
      }
}
```

Vous faites remarquer que la fonction `cibler_lieux()` semble être de complexité algorithmique $O(m \times n)$, où $m$ est `lieux.size()` et $n$ est `matous.size()`, et ce, en présumant que `pred()` ne fasse qu'un seul parcours de la séquence de félins (vous ne contrôlez pas ce comportement, alors mieux vaut y aller avec prudence). Si le nombre de félins croît et si le nombre de lieux est grand, il se peut que cette fonction soit coûteuse.

Vous savez que `pred()` sera suppléé par des spécialistes de la distribution d'animaux dans un territoire. Ces gens ont un bagage en mathématiques, mais on ne peut présumer qu'ils ont un bagage important d'aptitudes en optimisation de programmes. Cela signifie que si un gain d'échelle doit être obtenu, mieux vaut l'obtenir à même `cibler_lieux()`.

`Q00.0` – La première question est celle de la *possibilité*. **Pour `10` points**, serait-il possible de paralléliser `cibler_lieux()` à l'aide de plusieurs *threads*? Si oui, montrez comment. Sinon, expliquez comment l'algorithme devrait être modifié pour en arriver à une version qui, elle, pourrait être parallélisée.

`Q00.1` – La deuxième question est celle de la *pertinence*. **Pour `10` points**, que ce soit dans la version originale ou dans la version modifiée selon votre vision des choses (réf. : `Q00.0`), comment déterminer à quelle(s) condition(s) il sera pertinent de paralléliser l'algorithme décrit par `cibler_lieux()`?

## Q01 Que d'événements! (_____/20)

Votre apport à la résolution du problème exposé à `Q00` vous apporte une forme de respect de vos pairs. En particulier, votre collègue Bertrand qui apprend la programmation de systèmes parallèles et concurrents à partir d'un site populaire de sociosoutien (et qui trouve que la vie était bien plus simple avant l'avènement de programmes composés de multiples *threads*) vous tient désormais en haute estime.

D'ailleurs, à ce propos, Bertrand a un problème à savoir l'extrait de code suivant, qui fonctionne depuis longtemps, mais avec la croissance du nombre de félins à examiner (et la croissance probable du nombre de tests à réaliser par félin) se transforme graduellement en un bien vilain goulot d'étranglement :

```
class Bilan { /* ... */ };
class Matou { /* ... */ };
Bilan fusionner_bilans(Bilan, Bilan);
Bilan tester_puces(const Matou&);
Bilan tester_tiques(const Matou&);
Bilan tester_surpoids(const Matou&);
Bilan tester_rhino(const Matou&);
Bilan evaluer(const Matou &matou) {
   Bilan bilan = tester_puces(matou);
   bilan = fusionner_bilans(bilan, tester_tiques(matou));
   bilan = fusionner_bilans(bilan, tester_surpoids(matou));
   bilan = fusionner_bilans(bilan, tester_rhino(matou));
   return bilan;
}
class CanalMatou { /* ... */ };
Matou lire(CanalMatou&);
vector<pair<Matou,Bilan>> evaluer_sante(vector<CanalMatou>& canaux) {
   vector<pair<Matou,Bilan>> v;
   for(auto &canal : canaux) {
      Matou matou = lire(canal);
      Bilan bilan = evaluer(matou);
      v.emplace_back(make_pair(matou,bilan));
   }
   return v;
}
```

Bertrand cherche vos conseils pour accélérer le tout, dans l'espoir que vous puissiez l'aider à accroître le débit de traitement de cette section du programme.

Il vous explique brièvement comment fonctionne le logiciel existant :

- Un `CanalMatou` encapsule un mécanisme de communication (typiquement un *socket*) avec une micropuce sur un félin et doit être utilisé comme tel.
- La fonction `lire(CanalMatou&)` bloque sur le canal passé en paramètre jusqu'au moment où des informations sur le félin à l'autre extrémité du canal seront reçues. Chaque micropuce émet des données à un rythme variant, entre autres, selon les mouvements du félin en question, les envois sont donc à un rythme irrégulier d'un félin à l'autre. Cette fonction ne peut pas être modifiée et elle ne peut pas être remplacée pour le moment.
- La fonction `evaluer_sante()` doit produire un bilan de santé de tous les félins pour lesquels elle reçoit un `CanalMatou`.
- Enfin, la fonction `evaluer()` produit le bilan de santé d'un matou.

Bertand aimerait accélérer le calcul fait par `evaluer()`, car cette fonction est souvent appelée, mais note que le nombre de tests à réaliser pour produire un bilan tend à croître (il y en a quatre pour le moment, mais cette situation peut s'alourdir). Il aimerait aussi que la lecture des canaux faite dans `evaluer_sante()` soit plus efficace, car pour le moment, les informations sur les félins sont lues en séquence, ce qui fait que si le premier félin fait la sieste, il n'est pas possible de traiter les données des autres félins même si celles-ci auraient pu être rendues disponibles.

`Q01.0` – **Pour `10` points**, décrivez une approche pour paralléliser le traitement des tests réalisés par la fonction `evaluer()`. .

`Q01.1` – **Pour `10` points**, proposez une approche pour paralléliser à la fois la consommation des données et la production du vecteur qui contiendra les divers bilans de santé calculés par la fonction `evaluer_sante()` :

## Q02 Si la tendance se maintient… (_____/25)

La croissance de Minettes & matous et une éventuelle prise en charge des chiens errants par une nouvelle version du logiciel pose la question de l'échelonnabilité (*Scalability*) du système, du fait que sur un ordinateur donné, l'accroissement du nombre de threads finit par avoir raison des ressources disponibles; ajouter des ordinateurs à un réseau est une tâche relativement simple à réaliser et semble être la meilleure option pour survivre au fait que la charge de travail ne cesse de prendre de l'ampleur.

Le modèle `MPI`, qui a bonne réputation, a été choisi pour la répartition des calculs. Chaque processus recevra un ensemble de données sur les félins (et les chiens éventuellement), réalisera des calculs sur la base de ces données et renverra, le cas échéant, le fruit de ses calculs. Pour les fins du design initial de cette nouvelle version du système, votre équipe et vous-même essayez de mettre au point un système de prévision des endroits où s'amasseront les animaux selon l'heure de la journée, pour rendre plus efficaces les opérations de capture (*Catch & Rescue*).

Vous savez par expérience que certains calculs seront relativement courts, n'occupant un cœur que quelques secondes à la fois, alors que d'autres peuvent être beaucoup plus longs et prendre plusieurs minutes à se compléter. Le fait que même les calculs les plus courts occupent un cœur pour un temps de l'ordre de la seconde est un autre signe qu'une répartition du traitement sur plusieurs ordinateurs est une option raisonnable (le temps de transfert de données sera nettement inférieur au temps de calcul).

`Q02.0` – Supposons qu'un processus délègue du travail à $n$ autres processus, et que ce travail soit découpé en $m$ échantillons, où $m$ est nettement plus grand que $n$. **Pour `10` points**, proposez une stratégie pour tenir occupés au maximum les $n$ processus. Par « occupés », on entend que l'on souhaite que chaque processus soit utilisé à près de 100 % de sa capacité pour réaliser des tâches pertinentes au problème à résoudre.

`Q02.1` – **Pour `10` points**, proposez une stratégie pour éviter qu'un processus continue à travailler sur un calcul qui ne serait plus pertinent dû aux résultats des calculs faits par un autre processus.

`Q02.2` – Bien que votre projet utilise `MPI` pour répartir les calculs entre ordinateurs, vos collègues se demandent si `MPI` est le meilleur modèle pour répartir les calculs entre plusieurs processus sur un *même* ordinateur? **Pour `5` points**, décrivez une technologie différente de `MPI` pour répartir les calculs sur un même ordinateur, que ce soit à l'aide de plusieurs *threads* ou à l'aide de plusieurs processus et expliquez en quoi cette technologie se compare à `MPI`, que ce soit en mieux ou en pire.

## Q03 Choix et pédagogie ( _____/15)

Bertrand, impressionné par votre fine compréhension des enjeux du parallélisme et de la concurrence, profite de votre présence pour poser quelques questions qui le turlupinent depuis des lustres.

`Q03.0` – Bertrand vous dit avoir lu sur plusieurs tribunes qu'il serait sage, dans un programme concurrent, de privilégier les objets immuables lorsque ceux-ci sont partagés entre au moins deux *threads*. **Pour `5` points**, expliquez-lui ce dont il en retourne :_____

`Q03.1` – Bertrand se plaint que les conteneurs conventionnels (comme `std::vector` ou `std::list` si on prend l'exemple de C++) ne soient pas *Thread-Safe*, et vous demande pourquoi il en est ainsi. **Pour `5` points**, expliquez-lui de quoi il en retourne : _____

`Q03.2` – Bertrand a expérimenté avec des processus multiprogrammés à l'aide de *threads* lancés sur demande et parvient en général à bien s'en tirer. Il dit avoir eu vent de regroupements de *threads* (*Thread Pools*), mais ne comprend pas bien pourquoi on aurait recours à un tel mécanisme. **Pour `5` points**, donnez-lui un avantage de ce mécanisme sur une approche où les *threads* seraient lancés sur demande. _____

## Q04 Une fausse bonne nouvelle ( _____/10)

Bertrand est bien préoccupé. En effet, il vous montre les deux programmes suivants :

**Programme A**

```
long long fct() {
   enum { N = 10'000'000 };
   long long r = {};
   for (int i = 1; i <= N; ++i) {
      r += static_cast<int>(sqrt(i));
   }
   return r;
}
```

**Programme B**

```
long long fct() {
   enum { N = 10'000'000 };
   long long r0 = {};
   long long r1 = {};
   thread th0{ [&r0] {
      for (int i = 1; i < N / 2; ++i)
         r0 += static_cast<int>(sqrt(i));
   } };
   thread th1 { [&r1] {
      for (int i = N / 2; i <= N; ++i
         r1 += static_cast<int>(sqrt(i));
   } };
   th0.join(); th1.join();
   return r0 + r1;
}
```

La source de ses préoccupations est la suivante : il pensait à peu près couper le temps de calcul de moitié en passant du programme A au programme B. Cependant, en pratique, bien qu'il obtienne à l'occasion un gain de vitesse, il se trouve que le temps d'exécution du programme B est erratique et que, la plupart du temps, le programme B est *significativement* plus lent que le programme A. Bertrand est par contre content de faire remarquer que les deux fonctions retournent en tout temps la même valeur.

Quelle est selon vous la raison de ce comportement du programme B? Prenez soin d'expliquer votre réponse. _____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

## Q05 La clé du bonheur? ( _____/10)

Votre collègue Bertrand dit avoir écrit une classe Java pleinement protégée contre les problèmes résultant d'accès concurrents et vous la présente fièrement :

```
class PileSynchroDeBertrand {
   private class Noeud {
      public final String valeur;
      public Noeud pred;
      public Noeud(String valeur) {
         this.valeur = valeur;
         pred = null;
      }
   }
   private void attendreVerrou() {
      while(verrou) { // attendre activement que le verrou soit disponible
      }
   }
   private boolean verrou;
   private Noeud tete;
   public PileSynchroDeBertrand() {
      tete = null;
      verrou = false;
   }
   public boolean isEmpty() {
      return tete == null;
   }
   public void push(String valeur) {
      attendreVerrou();
      verrou = true;
      Noeud p = new Noeud(valeur);
      p.pred = tete;
      tete = P;
      verrou = false;
   }
   public String pop() {
      attendreVerrou();
      verrou = true;
      if (isEmpty()) {
          throw new Exception("Pile de Bertrand vide");
      }
      String valeur = tete.valeur;
      tete = tete.pred;
      verrou = false;
   }
}
```

Bertrand est-t-il en droit d'être fier? Si oui, expliquez pourquoi. Sinon, dites pourquoi et appuyez votre réponse d'un exemple. _____

## Q06 Pourtant, tout semblait si beau… ( _____/10)

Le plus récent projet de Minettes & matous doit comprendre un point de rendez-vous modélisé, ci-dessous, par une ébauche produite par un collègue non identifié (vous suspectez Bertrand) :

```
class RendezVous {
   atomic<int> combien;
   vector<function<void()>> fct;
   mutex m;
   void ajouter(function<void()> && f) {
      lock_guard<mutex> _ { m };
      fct.emplace_back(std::move(f));
   }
public:
   RendezVous(int n) : combien{ n } {
   }
   void attendre(function<void()> && f) {
      ajouter(std::move(f));
      --combien;
      if (combien == 0) {
         for(auto & f : fct) f();
      }
   }
};
```

L'idée de cette classe, qui compile et s'exécute sans heurts pour l'essentiel, va comme suit :

- À la construction, un `RendezVous` est initialisé avec le nombre de fonctions qui seront mises en attente.
- À chaque fois qu'on appelle sa méthode `attendre()`, une fonction est ajoutée à son vecteur `fct` et le nombre de fonctions à attendre (variable `combien`) est décrémenté.
- Lorsque le compteur tombe à zéro, toutes les fonctions sont arrivées au point de rendez-vous et le `RendezVous` exécute successivement toutes les fonctions.

Le tout fonctionne assez bien dans l'ensemble, même quand un `RendezVous` donné est utilisé concurremment par plusieurs *threads*. Les tests montrent que jamais, dans votre programme, un `RendezVous` n'est utilisé par un trop grand nombre de *threads*. Toutefois, à l'occasion, les fonctions d'un même `RendezVous` sont appelées plusieurs fois, ce qui laisse vos collègues quelque peu perplexes.

**Pour 5 points**, quelle est la raison de cet étrange comportement? Et **pour 5 points**, expliquez comment corriger la situation : _____

_____

_____

_____

_____

_____

## Annexe – détails techniques

À titre de rappel, voici les détails techniques qui apparaissent dans cet examen (pour vous éviter des maux de tête si vous avez un trou de mémoire).

| Question(s) | Détail |
|---|---|
| Général | Les itérateurs en C++ sont manipulés par paires, définissant ce qu'on nomme une séquence standard définie sur un intervalle à demi ouvert $[debut, fin[$. |
| `Q00` | La fonction `begin(v)` retourne un itérateur sur le début de `v`. |
| `Q00` | La fonction `end(v)` retourne un itérateur sur la fin de `v`. |
| `Q00` | Un prédicat est une opération booléenne. |
| `Q00`, `Q01`, `Q06` | La notation `for(const auto &e :v)` signifie « pour chaque élément `e` dans `v`, pris par référence-vers-`const` ». Le `const` est omis si `e` doit être modifié par la répétitive. |
| `Q00`, `Q01`, `Q06` | La méthode `emplace_back(args)` d'un `vector<T>` construit un `T` à la fin du conteneur en passant les paramètres `args` à son constructeur, accroissant sa capacité au passage s'il est plein. |
| `Q01` | Le type `pair<K,V>` contient deux valeurs, soit `first`, de type `K`, et `second`, de type `V`. |
| `Q06` | Un `function<T(args)>` encapsule une opération prenant `args` en paramètre et retournant un `T`; un `function<void()>` ne prend donc pas de paramètres et ne retourne rien. |