420KH2 – TP « bonbon »

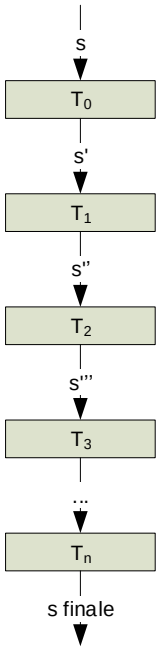
Ce travail pratique vous amènera à mettre en pratique des techniques pour gérer la concurrence et la synchronisation dans un programme. Vous mandat pour ce travail pratique sera d'implémenter un schéma de conception typique du parallélisme, soit le **pipeline**.

Concrètement, vous devrez lire plusieurs objets, un après l'autre, et leur appliquer une séquence de transformations simples menant, en fin de parcours, à un objet pleinement transformé.

Pour les besoins de ce travail, votre pipeline devra être générique (`Pipeline<T>` pour un certain type `T`) et contenir des opérations (fonctions, foncteurs, λ, etc.) qui consomment un `T` et retournent un `T`.

Structure d'un pipeline

Un pipeline est une séquence d'opérations appliquées à un élément. Chaque étape du pipeline s'exécute concurremment avec les autres, mais alors que l'étape T_0 opère sur la donnée n, l'étape T_1 opère quant à elle sur la donnée $n-1$, l'étape T_2 opère sur la données $n-2$, etc.



Chaque étape d'un pipeline est typiquement un fil d'exécution (un *thread*) qui :

- Consomme un `T` d'une file en entrée
- Passe ce `T` en paramètre à une fonction
- Prend la valeur de retour de cette fonction et l'insère dans une file en sortie

Notez que la sortie d'une étape du pipeline constitue aussi l'entrée de l'étape suivante. Pour cette raison, ces étapes sont des zones de transit qui doivent être synchronisées.

Initialement, un pipeline est vide. Quand on le nourrit de données à traiter, il se remplit peu à peu, et est typiquement au maximum de son efficacité lorsque toutes ses étapes sont alimentées.

Par nature, un pipeline est aussi rapide que la plus lente de ses étapes (un peu comme le veut l'adage à l'effet qu'une chaîne est aussi fragile que l'est le plus faible de ses maillons).

Programme de test (semi) imposé

Pour ce travail « bonbon », le programme de test sera « semi-imposé ». Par « semi-imposé », j'entends que l'interface d'un `Pipeline<T>` doit respecter au moins celle utilisée dans le programme principal suivant, mais que vous pouvez tester votre implémentation avec d'autres types `T` et avec d'autres étapes / d'autres données.

Le code « semi-imposé » suit (je présume ici un fichier nommé `Principal.cpp`; ajustez au besoin, en particulier si vous avez utilisé un autre nom de fichier) :

```
// ...
std::string operator "" _file(const char* s, std::size_t) {
   std::ifstream in{ s };
   return { std::istreambuf_iterator<char>{ in }, std::istreambuf_iterator<char>{} };
}
int main() {
   // ... dans ce qui suit, vous pouvez ajouter le code en commentaires pour vous divertir
   using namespace std;
   Pipeline<string> pipeline;
   pipeline.add_step([](string s) {
      return /*std::reverse(begin(s), end(s)),*/ s;
   });
   pipeline.add_step([loc = locale{ "" }](string s) {
      return /*std::transform(begin(s), end(s), begin(s), [&loc](char c) {
         return toupper(c, loc);
      }),*/ s;
   });
   pipeline.add_step([](string s) {
      cout << s << endl; return string{};
   });
   pipeline.feed("J'aime mon prof"s, "Yo"s, "Eh ben"s, "Principal.cpp"_file);
   pipeline.start();
   char c;
   cin.get(c);
   pipeline.stop();
}
```

Le code de test suppose ce qui suit :

- Les appels à `add_step()` doivent être faits avant que `start()` ne soit appelé (vous pouvez lever une exception si le code client contrevient à cette règle). La raison pour ceci est que `add_step()` doit à la fois ajouter une étape au traitement et une zone de transit menant vers l'étape suivante, alors il serait déraisonnable de permettre ceci une fois le traitement du pipeline en action
- Vous pouvez créer des fils d'exécution à loisir, mais ceux-ci ne doivent pas commencer à traiter les données avant que `start()` ne soit appelé
- Vous pouvez ajouter des données à traiter (`feed()`) quand bon vous semblera, et passer à cette fonction autant de données que vous ne le souhaiterez

Remarques complémentaires

Bien que ce travail puisse être réalisé sans traits, vous pouvez obtenir de meilleurs messages d'erreurs si jamais quelqu'un cherche à insérer une étape qui n'a pas la bonne signature dans votre pipeline à l'aide de traits et d'assertions statiques.

Par exemple, avec un `Pipeline<string>`, l'opération suivante :

```
pipeline.add_step([](string) { return 3; });
```

… ne devrait pas compiler (elle retourne un `int` mais devrait retourner un `string`), mais vous pouvez générer un message d'erreur à votre convenance à l'aide de ces techniques. Je vous invite à explorer :

- L'opérateur `decltype`
- La « fonction » `declval<T>()`, et
- Le trait `is_convertible`

… pour développer une stratégie en ce sens.

Spécifications techniques

Ce travail se veut agréable. Il est, tout d'abord, totalement optionnel, mais devrait être divertissant et, ce qui n'est pas négligeable, plutôt utile! Si vous choisissez de le faire, assurez-vous de réfléchir, de mettre en application les techniques vues en classe, et de bavarder avec votre chic prof si vous êtes coincé(e).

À titre d'information, dans mon implémentation personnelle, j'ai utilisé :

- De la synchronisation (`mutex`, `atomic`, `condition_variable`, `lock_guard`, `unique_lock`)
- De la multiprogrammation (`thread`)
- Des contraintes de temps (`chrono`)
- Le mouvement
- Des λ
- Des pointeurs intelligents (`unique_ptr`)
- Des conteneurs (`vector`, `deque`), des algorithmes, des itérateurs
- Des énumérations fortes (`enum class`)
- Des boucles `for` modernes
- Des insertions par emplacement (`emplace_back()`)
- Un *template* variadique (pour `feed()`); en fait, j'ai utilisé une *Fold Expression*

… et ainsi de suite (ne vous en faites pas si votre implémentation est différente; je vous indique cela pour vous donner un aperçu de ce que j'ai utilisé). Ça fait une belle révision.

À titre indicatif, ces outils m'ont permis d'implémenter le pipeline au complet (incluant les classes auxiliaires que je me suis écrit pour que ce soit propre et compréhensible) en `86` lignes de code au total (sans compter les inclusions et les lignes vides, et sans compter le code de test).

Contraintes humaines et échéances

Organisation humaine	Individuel, t'sais
Format de la remise	Imprimé. Ne m'imprimez pas mon propre code; c'est les sources résultant de vos efforts que je veux lire!
Date de remise	Au plus tard au début du cours du mardi 16 mai 2019.
Impact sur la note	Si vous réussissez ce travail, sa note remplacera celle du moins bon travail pratique de votre session (ou celle d'un minitest mal réussi, si cela vous avantage plus). Je ferai ce changement manuellement dans Colnet, par contre (l'outil manque un peu de flexibilité pour de tels ajustements)

Amusez-vous bien!