UGA - IUT2 - Département informatique
BUT Semestre 6 - Programmation Concurrente et Répartie - 2024/2025

TP01 : Threads

0. Conseils, Rappels

Récupérer les fichiers à compléter

Vérifiez que vous avez bien installés les paquets suivants, en exécutant la commande (en étant root):

```
apt update && apt install -y build-essential clang libboost-all-dev
```

N'oubliez pas de faire les upgrade ce soir chez vous avec `apt upgrade` **(mais pas en début de TP !)**.

Les fichiers pour le TP sont sur chamilo, dans le dossier Documents, dans un fichier compressé pour le tp01 et le tp02. Les deux tps sont en lien, sur deux séances et partagent des fichiers de l'un à l'autre.

Commenter ses programmes

Commenter systématiquement vos programmes afin d'être en mesure, lorsque vous les relirez plus tard, de comprendre rapidement ce qu'ils faisaient.

Compilation et exécutables

Les `Makefile` fournis compilent vos programmes avec deux compilateurs :

- `clang` qui proposent des messages d'erreurs explicites;
- `gcc` qui produit des exécutables plus performants que `clang`.

Chaque compilateur crée les fichiers objets et exécutables dans un répertoire qui lui est propre : `bin-clang/` et `bin-gcc/` respectivement. Pour tester vos exécutables, il vous faudra donc taper :

```
bin-gcc/mon_executable
```

au lieu de

```
./mon_executable
```

1. Fonction ou Classe avec opérateur () ?

Se positionner dans le répertoire `tp01/fonction-ou-classe/`.

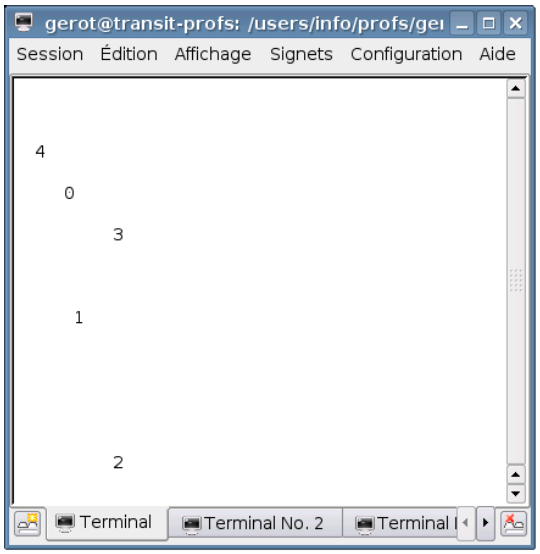
Le premier argument du constructeur **`std::thread()`** est un objet appelable qui peut être :

1. une fonction,
2. ou un objet, instance d'une classe munie d'un opérateur **`()`**.

Dorénavant, nous opterons toujours pour le deuxième choix. Pour bien le comprendre, il vous est demandé de traduire le programme `exemple_fonction.cpp` où les threads sont créés avec une fonction, en un programme équivalent mais où les threads sont créés avec des instances de la classe **`MonThread`** possédant un opérateur **`()`**. Pour cela, compléter les fichiers suivants :

- `exemple_classe.cpp`
- `MonThread.hpp`
- `MonThread.cpp`

2. Androïdes gérés par des threads



Dans cet exercice, il s'agit de créer des threads qui vont simuler des androïdes inspectant la surface d'une planète inconnue. Chaque thread devra calculer les mouvements de l'androïde qu'il représente et tracer dans votre terminal un caractère qui va reproduire ses mouvements (voir ci-contre).

La stratégie d'exploration adoptée est le mouvement brownien : les déplacements seront aléatoires. Un autre membre de l'équipe a déjà écrit la méthode **`marche_aleatoire()`** de la classe **`Androide`**, que devra exécuter chacun de vos threads. Cette méthode utilise deux attributs de la classe :

- un entier qui identifie l'androïde; c'est ce numéro identifiant que **`marche_aleatoire()`** va écrire dans votre terminal pour représenter l'androïde modélisé par le thread.
- un entier qui détermine sa vitesse de déplacement; c'est le temps de latence entre deux mouvements de l'androïde.

A la fin de son exploration, chaque numéro représentant un androïde s'immobilise et écrit une étoile à sa droite.

Enfin, les androïdes peuvent être classés en deux catégories : les androïdes rapides (latence faible entre deux mouvements), et les androïdes lents (latence élevée). La vitesse effective dépendra de chaque expérience, aléatoirement. En revanche, après un moment, le thread principal (simulant le QG de contrôle) diminue les deux latences et les androïdes devront accélerer en conséquence.

Instructions et Questions :

1. Se positionner dans le répertoire `tp01/androides/`.
2. Éditer `Androide.hpp` et `Androide.cpp` pour modifier le constructeur afin qu'il prenne en compte les arguments suivants :
   - le numéro **`numero`** de l'androïde;
   - sa latence **`latence`** : cette valeur détermine la vitesse des androïdes;
   - l'objet **`ecran`** utilisé pour les affichages.

   Une fois cette modification effectuée, le `numero` et la `latence` utilisés dans la méthode `marche_aléatoire` doivent être ceux passés en argument (il faut donc enlever leur définition locale).
3. Éditer `Androide.hpp` et `Androide.cpp` pour ajouter un opérateur **`()`** qui exécute la méthode **`marche_aleatoire()`**.
4. Compléter le programme principal pour :
   - créer **`nb_threads`** threads qui exécuteront l'opérateur **`()`** de la classe **`Androide`** via une instance anonyme de cette classe, utilisant les arguments suivants :
     - le numéro **`i`** de l'androïde;
     - sa latence **`latences[i%nb_latences]`** : les threads pairs simuleront les androïdes rapides et les impairs simuleront les lents;
     - l'objet **`ecran`** utilisé pour les affichages.
   - attendre la fin des threads avant de mettre fin au programme.
5. Compiler et tester votre programme pour vous assurer que **`nb_threads`** threads ont bien été créés. Exécuter dans un autre terminal la commande `ps -U NOM_DE_LOGIN -L -o pid,lwp,state,cmd` et retrouver les threads ainsi créés.
6. Vérifier également que les threads accélèrent au bout d'un moment. Indice : l'accélération est produite par un changement de la latence dans le thread principal. Si ce n'est pas déjà le cas, pensez à la copie de la latence qui est faite lors du lancement des threads, et remplacez cette copie par une référence.

3. Traitement multi-thread d'une image

Les ordinateurs actuels possèdent plusieurs processeurs, et chaque processeur possède plusieurs cœurs. Il est alors possible d'accelérer un calcul en exploitant ce parallélisme matériel.

Nous vous proposons de paralléliser une partie d'un algorithme de traitement d'une image pour compresser un flux vidéo au format MPEG. Cette compression consiste à représenter une seule fois les petites parties (blocs) de deux images successives. Nous avons donc besoin d'un algorithme qui recherche la position d'une petite image à l'intérieur d'une grande image. Les images sont stockées dans le format PPM. Vous disposez de la librairie `imagePPM.hpp` pour manipuler les images dans ce format. Cette librairie vous propose les éléments suivants :

- **`t_image`** : le type image "bitmap".
- **`read_image()`** : lecture d'un fichier image au format PPM dans un objet de type **`t_image`**
- **`write_image()`** : écriture d'une image de type **`t_image`** dans un fichier au format PPM
- **`search_image()`** : recherche une image dans une autre, retourne la somme des différences en valeur absolue de chaque pixel de l'image à rechercher. Le résultat est d'autant plus petit que l'image est proche. La valeur zéro dénote une correspondance parfaite.
- **`split_image()`** : découpe une image en deux sous-images avec un recouvrement de n pixels.

Instructions et Questions :

1. Se positionner dans le répertoire `tp01/recherche-image/`.
2. Compiler le programme `rech_img.cpp` et examiner les images `commun/data/grenoble.ppm` et `commun/data/IUT.ppm` par exemple avec la commande `eog` ou tout autre commande permettant d'afficher des images.
3. Exécuter le programme avec en paramètres l'image source et l'image recherchée en mesurant son temps d'exécution à l'aide de la commande `time`.

Temps CPU (user+sys)	Temps horloge (real)
.	.

   Examiner le résultat dans le fichier `/tmp/resultat-<votre_login>.ppm`. La position de l'image retrouvée est indiquée par un rectangle rouge.

Vous devez maintenant écrire une version multi-thread de cette recherche d'images. Cette version incomplète est dans le fichier `rech_img_thread.cpp`. L'image source doit être découpée en plusieurs sous-images sur lesquelles chaque thread réalisera sa recherche. La fonction **`split_image()`** réalise le découpage d'une image en deux. Chaque thread doit alors traiter son image et rendre son résultat grâce à un objet de la classe **`SearchImageData`**.

Vous devez compléter le code de la manière suivante :

1. Réaliser la boucle de création des threads exécutant l'opérateur **`()`** d'instances anonymes de la classe **`Chercheur`** créées avec les bons paramètres (sous-partie de l'image où effectuer la recherche, imagette recherchée, et attributs de l'objet de la classe **`SearchImageData`** associé au thread, où celui-ci rangera les resultats de sa recherche).
2. Compléter en conséquence `Chercheur.hpp` et `Chercheur.cpp`. En particulier, chaque thread doit exécuter la fonction **`search_image()`** en lui passant les bons paramètres.
3. Réaliser la boucle d'attente de la fin du travail des threads.
4. Finalement réaliser la boucle de recherche de la meilleure solution trouvée par chaque thread.
5. Tester votre programme pour obtenir la même solution que dans la version non parallélisée. Constater avec la commande `time` le gain de temps de calcul réalisé par le parallélisme.

nb threads	Temps CPU (user+sys)	Temps horloge (real)
2	.	.

   Comment expliquez-vous la valeur du temps CPU ?

6. Tester le programme, toujours avec `time`, mais cette fois-ci avec 4 puis 8 threads, chaque thread effectuant la recherche sur un quart ou un huitième de l'image, respectivement.

nb threads	Temps CPU (user+sys)	Temps horloge (real)
4	.	.
8	.	.

   Comment expliquez-vous ce résultat ? Donnez votre résultat à votre enseignant, ainsi que le nombre de coeurs de calcul donné par la commande `lscpu`.

4. Processus ou thread ?

Nous avons vu en cours les différences théoriques et pratiques qui peuvent exister entre un processus et un thread. Cet exercice fait un bilan pratique.

Se positionner dans le répertoire `tp01/processus-ou-thread/`.

4.1. Appels systèmes vs. méthodes de la classe `std::thread`

Instructions et Questions :

1. Exécuter la commande `meld thread.cpp processus.cpp` et observer la différence entre les deux codes.
2. Compléter le tableau suivant résumant les types et procédures associés aux processus et threads.

nature	type de données / classe	procédure de création / constructeur	procédure / méthode d'attente de fin d'exécution
processus			
thread			

3. Compiler ces programmes et exécuter. Pourquoi les deux programmes ne rendent-ils pas le même résultat ?

.

4.2. Zones mémoire

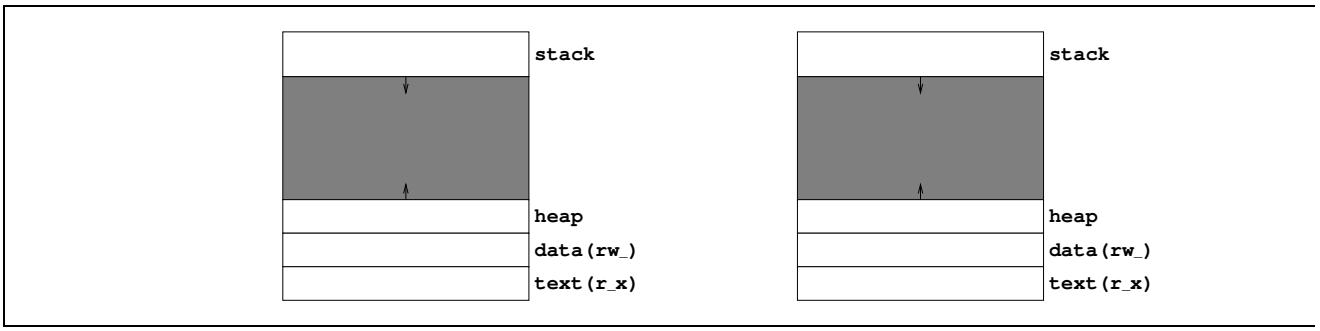
Dans cet exercice, nous rappelons que, contrairement à un processus père et son processus fils, les threads d'un même processus partagent le même espace mémoire. En revanche, chaque thread possède sa propre pile.

Instructions et Questions :

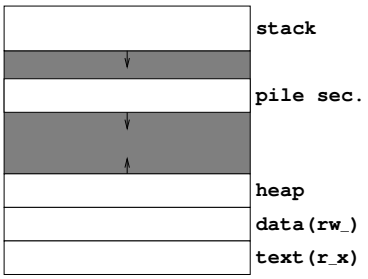
1. Lire le contenu des fichiers **zones_mem_processus.cpp** et **zones_mem_thread.cpp** et repérer les trois variables suivantes :
   - **`var_globale`**
   - **`var_pile_main`**
   - **`var_pile_fonc`**
2. Dans un terminal aux dimensions maximisées, compiler **zones_mem_processus.cpp** et **zones_mem_thread.cpp**, puis exécuter les binaires ainsi générés. Compléter les figures ci-dessous en plaçant les variables suivantes précédées des premiers chiffres de leurs adresses, dans les bons segments :
   - **`var_globale`** (processus père ou fils)
   - **`var_pile_main`** (processus père ou fils)
   - **`var_pile_fonc`** (processus fils ou thread secondaire)
   - **`var_pile_fonc`** (processus père ou thread principal)

   Nous rappelons que l'ordre des segments affichés est inversé par rapport aux figures.

Exécution de zones_mem_processus



Exécution de zones_mem_thread



4.3. Plusieurs threads pour un même processus

Dans cet exercice nous mettons en évidence la possibilité de définir plusieurs exécutions simultanées (tâches) pour un même programme. La tâche principale est le processus qui exécute la procédure **`main()`**. Les tâches secondaires sont créées grâce au constructeur de la classe **`std::thread`**.

Noter que l'identifiant de chaque objet de type **`std::thread`** donné par la bibliothèque standard C++ (obtenu par **`get_id()`**) est différent de celui que le système d'exploitation donne à l'exécution qui lui est associée (obtenu par **`syscall(SYS_gettid)`**).

Instructions et Questions :

1. Compléter **simple_thread.cpp** pour créer un thread **`mon_thread`** qui exécute **`ma_fonction()`**. Compiler et exécuter. Qui de la tâche principale ou secondaire a écrit chacune des deux lignes ?
2. Compléter **multiples_threads.cpp** pour créer **`nb_threads`** threads qui seront stockés dans le vecteur **`mes_threads`**. Chacun de ces threads doit exécuter **`ma_fonction()`**. Compiler et exécuter. Expliquer les différences et similitudes entre les différentes lignes qui s'affichent dans le terminal. *Remarque : les lignes affichées par les différents threads peuvent d'entremêler. La résolution de ce problème d'accès concurrent à la sortie standard sera l'objet du prochain chapitre.*