

TP2 INF2610

Communication interprocessus et Synchronisation

Groupe 02L

Polytechnique Montréal

Hiver 2025

Date de remise: Voir le site Moodle du cours (section TP2)

Pondération: 10%

Ce travail va consister de **trois parties**. Si vous comprenez bien la matière du cours, vous devriez être en mesure de compléter le travail de la manière suivante :

Échéancier

Partie	Échéancier
Partie 1	Séance de laboratoire #3
Partie 2	Séance de laboratoire #4
Partie 3	À terminer chez soi

Répartition des points

Partie	Description	Points
Partie 1	Communication interprocessus	6
Partie 2	Synchronisation	6
Partie 3	Synthèse de la matière	8
Total		20

Des informations additionnelles pour la correction de chaque partie sont disponibles dans l'énoncé.

Partie 1

Objectifs

Cette partie du travail pratique (TP) a pour but de vous familiariser avec les tubes anonymes et nommés, utilisés pour faire communiquer des processus. À l'issue de cette partie, vous serez capable de:

- Faire communiquer des processus au moyens de tubes de communication anonymes et nommés.
- Rediriger les flux d'entrée et de sortie d'un processus vers des fichiers et des tubes de communication.

Cette partie du TP est composé de deux questions indépendantes que vous pouvez traiter séparément.


Question 1 - Tubes anonymes et redirections

L'objectif de cette question est de reproduire en langage C le traitement réalisé par la composition de commandes suivante:

```
rev < In.txt | rev | diff - In.txt -s
```

 Où:

- `In.txt` est un fichier fourni contenant le text à inverser;
- `rev` et `diff` sont des fichiers exécutables dont les chemins d'accès sont dans la variable d'environnement `PATH`. La commande `rev In.txt`, sans paramètres, inverse l'ordre des caractères de chaque ligne lue à partir de son entrée standard. Le résultat de cette inversion est affiché sur la sortie standard. La commande `diff - In.txt -s` se charge de comparer le fichier lu à partir de l'entrée standard avec le fichier `In.txt`. Le résultat de cette comparaison est affiché sur la sortie standard. Dans le cas de la composition de commandes ci-dessus, l'affichage devrait indiquer l'identité des fichiers comparés.
- Les opérateurs "`<`" et "`>`" permettent respectivement de rediriger l'entrée et la sortie standards vers des fichiers.

 Complétez le code dans le fichier ***TubesAnonymes.c*** afin de réaliser le même traitement que la ligne de commandes ci-dessus. Vous devez également prendre en compte les exigences suivantes:

- Les trois processus que vous allez créer pour exécuter les commandes simples `rev`, `rev` et `diff` doivent être des processus fils du processus principal.
- Les descripteurs de fichiers non utiles doivent être fermés.
- Chaque processus père qui ne se transforme pas doit attendre la fin de ses fils avant de se terminer.
- Il n'est pas demandé de traiter les erreurs. Par contre, en cas de doute sur le bon fonctionnement de votre programme, n'hésitez pas à tester les valeurs de retour de vos appels système.

Question 2 - Tubes nommés et redirections

Dans cette question, vous devez reproduire le même traitement mais en utilisant un tube nommé au lieu d'un tube anonyme:

```
rev < In.txt | rev | diff - In.txt -s
```

Le code à compléter se trouve dans le fichier ***TubesNommes.c***.

Compilation, exécution et remise

Pour compiler et exécuter le cette partie du TP, lancez successivement les commandes `make` et `./CommLab` dans le répertoire de la partie.

Lancez la commande `make handin` dans le répertoire de la partie du TP afin de créer l'archive `handin.tar.gz`.

Barème

Question	Description	Points
1	Création et transformation des processus	4
	Tubes et redirections des E/S standards	6
	Fermeture des descripteurs de fichiers et attente de fin des processus	2
	Résultat correct	2
2	Communication par tubes nommés	4
	Clarté du code et commentaires	2
Total		20

Partie 2

Objectifs

Cette partie du TP a pour but de vous familiariser avec les threads et les mécanismes de synchronisation de l'API *POSIX*. Il s'agit de compléter le programme `GuerreDesChiffres.c` suivant, pour créer et synchroniser des threads selon le modèle de synchronisation producteurs et consommateurs. Dans ce modèle, il y a un ensemble de threads producteurs et un ensemble de threads consommateurs qui communiquent via un tampon de taille limitée. Les nombres de threads producteurs et de threads consommateurs ainsi que la taille du tampon sont des paramètres du programme `GuerreDesChiffres.c`.

Comportement attendu du programme

Les threads producteurs et consommateurs sont créés dans la fonction `main` du programme (thread principal). Ces threads ont chacun un seul paramètre qui correspond à son numéro: 0 est le numéro du premier producteur/consommateur créé, 1 est le numéro du second producteur/consommateur créé, etc.

Chaque thread producteur consiste en une boucle sans fin. À chaque itération, il génère aléatoirement un chiffre non nul qu'il dépose dans le tampon. Il calcule également la somme des chiffres générés. Cette somme est communiquée au thread principal via `pthread_exit` et `pthread_join`. Les threads producteurs mémorisent aussi dans une variable globale le nombre de chiffres produits par l'ensemble des producteurs.

De façon similaire, chaque thread consommateur est composé d'une boucle sans fin. À chaque itération, il récupère du tampon un chiffre. Il calcule également la somme des chiffres récupérés. Cette somme est communiquée au thread principal via `pthread_exit` et `pthread_join`. Le nombre de chiffres consommés par tous les threads consommateurs est mémorisé dans une autre variable globale.

Après la création de tous les threads, le thread principal du programme arme une alarme de 1 seconde (en utilisant l'appel système `alarm`) puis se met en attente de la fin de tous les threads producteurs. Le programme doit donc faire le nécessaire pour capter le signal `SIGALRM`. Le traitement du signal consiste à mettre à `true` une variable globale `flag_de_fin` qui est initialisée à `false`.

À la fin de tous les threads producteurs, le thread principal dépose dans le tampon autant de chiffres 0 qu'il y a de threads consommateurs puis se met en attente de tous les threads consommateurs. Il affiche ensuite les sommes des chiffres produits et consommés. Ces deux sommes devraient être égales. Il affiche également les nombres de chiffres produits par les producteurs et consommés par les consommateurs avant de se terminer. Ces deux nombres devraient aussi être égaux.

En plus du traitement décrit ci-dessus, chaque thread producteur teste, à la fin de chaque itération, la valeur de la variable `flag_de_fin`. Il met fin à son traitement, si cette valeur est `true`. Chaque thread consommateur met fin à son traitement lorsqu'il récupérera le chiffre 0 du tampon. Il doit cependant compléter l'itération en cours avant de se terminer.

Listing 1: Programme GuerreDesChiffres.c à compléter

```
// ...

// fonction exécutée par les producteurs
void* producteur( void* pid) {
    //...
    while(1) { // ...
        /* générer aléatoirement un chiffre non nul à déposer dans le
           tampon. Vous pouvez utiliser : srand(time(NULL));
           (à appeler une seule fois) et x=(rand()%9) + 1;
           pour générer aléatoirement un chiffre dans x. */
        // ...
    }
    // ...
}

// fonction exécutée par les consommateurs
void* consommateur(void *cid) {
    // ...
    while(1) { // ...
        // retirer un chiffre du tampon.
        // ...
    }
    // ...
}

// fonction main
int main(int argc, char* argv[]) {
    /* Les paramètres du programme sont, dans l'ordre :
       le nombre de producteurs, le nombre de consommateurs
       et la taille du tampon.*/
    // ....
}
```

Travail à faire et remise

Il vous est demandé de compléter le programme `GuerreDesChiffres.c` afin qu'il réalise le traitement attendu décrit ci-dessus. Faites attention aux accès concurrents et aux interblocages.

Utilisez les sémaphores *POSIX* (`sem_wait`, `sem_post`, `sem_init`, etc.), pour éviter les conditions de concurrence et synchroniser adéquatement tous les threads *POSIX* créés. Indiquez, sous forme de commentaires dans le code, le rôle de chaque sémaphore. Attention à l'utilisation abusive de sémaphores.

Compilez votre programme en utilisant la ligne de commande :

```
gcc -pthread GuerreDesChiffres.c -o GuerreDesChiffres
```

Testez votre programme pour différentes valeurs de paramètres. Par exemple, pour le tester pour le cas de 3 producteurs, 2 consommateurs et un tampon de taille 5, tapez la ligne de commande suivante :

```
./GuerreDesChiffres 3 2 5
```

En cas de doute sur le bon fonctionnement de votre programme, n'hésitez pas à tester les valeurs de retours de vos appels système et variables.

Pour la remise, déposez le fichier contenant votre programme sur le site *Moodle* du cours.

Barème

Description	Points
Gestion des paramètres, création du tampon et des threads	4
Synchronisation adéquate des threads	5
Signal, attente, communication et terminaison des threads	5
Bon fonctionnement du code	4
Clarté du code et commentaires	2
Total	20

Partie 3

Objectifs

Cette section vise à vous permettre d'appliquer les connaissances acquises durant le travail pratique.

Vos solutions devraient réutiliser les concepts présentés aux parties antérieures du TP.

Question 1 : Merge-sort parallèle

Les ordinateurs modernes sont dotés de plusieurs coeurs pour effectuer des calculs parallèles. Nous voulons prendre avantage de ces ressources pour améliorer la performance de notre tri.

Dans le fichier `mergesort.c`, vous avez accès au code de l'algorithme en C. On vous demande de modifier le programme de manière qu'il soit capable de respecter les critères suivants :

- Le programme doit accepter deux arguments. Le premier argument correspond au nombre d'éléments X à trier dans le tableau. Le deuxième correspond au nombre de processus N .
- Vous devez créer N processus qui vont se partager en parallèle l'exécution de votre fonction de tri.
- Lorsqu'un processus termine son exécution, il doit communiquer avec le processus parent et accéder à la mémoire partagée afin de placer les éléments triés.

Votre code sera évalué selon son comportement ainsi que son degré de mobilisation de la matière du cours. Si votre programme ne réutilise pas d'éléments présentés aux parties 1 et 2, aucun point ne sera accordé.

Questions de compréhension :

Pour chaque question, donnez une explication courte. **1-5 lignes max.** Écrivez vos réponses dans le fichier `answers.txt`.

1. Supposez que vous faites passer le nombre de processus de N à $N + \delta$, où N correspond au nombre de coeurs disponibles. Quel sera l'impact de ce changement sur la performance de l'algorithme?
2. Nous utilisons un tableau comme structure de données. Quel autre type de structure pourrait être utilisé?

Question 2 : Journalisation du merge-sort

Pour valider le comportement de votre programme, on vous demande de créer un système qui peut journaliser ("logger") l'opération de tri. Dans le même fichier (*merge_sort.c*), on souhaite retrouver les éléments suivants:

- À chaque fois qu'un tri se termine, l'index de début, l'index de fin et les éléments triés seront envoyés au fichier *sorted_array.txt*. Vous devez afficher le tableau avant le tri dans le fichier. Par exemple :
array = [2,3,1,0,21,19,70,43,17]
Start = 0, End = 1, sorted = [2,3]
Start = 2, End = 3, sorted = [0,1]
Start = 0, End = 3, sorted = [0,1,2,3]
- Le temps d'exécution de la fonction de tri doit être chronométré. Vous devez utiliser la fonction *gettimeofday* pour mesurer le temps d'exécution.

Questions de compréhension :

Pour chaque question, donnez une explication courte. **1-5 lignes max.** Ajoutez vos réponses au fichier *answers.txt*.

1. Générez des résultats de temps d'exécution de votre algorithme parallèle avec journalisation et de l'algorithme *merge sort* traditionnel (sans parallélisme ni de journalisation). Assurez-vous que les résultats soient représentatifs de la réalité, sinon vos réponses aux questions suivantes ne seront pas évaluées. Vous devez générer des résultats pour les valeurs suivantes du nombre *N* de processus :

$$N = 1, 4, 8, 16$$

2. Comparativement au programme initial, votre programme est **combien de fois plus rapide ou lent**?
3. Ce résultat vous surprend-il? Justifiez.
4. Si vous mettez en commentaires ("*comment out*") le code de journalisation, le programme est-il plus rapide ou lent? Quel est l'ordre de grandeur du changement en performance?
5. Proposez **deux** approches pour améliorer la performance. Dans le fichier *merge_sort_upgrade.c* implémentez **l'une** des solutions que vous proposez. **Quel est l'ordre de grandeur de l'amélioration observée de la performance?** Attention, proposer de retirer la journalisation n'est pas une approche acceptée.

Remise

Les fichiers *merge_sort.c*, *merge_sort_upgrade.c*, *sorted_array.txt* et *answers.txt* doivent être remis pour évaluation.

Barème

Description	Points
Implémentation - Merge-sort parallèle	7
Questions de compréhension - Merge-sort parallèle	3
Implémentation - Journalisation	3
Questions de compréhension - Journalisation	5
Lisibilité du code	2
Total	20

**Si vous ne complétez pas le code,
aucun point ne vous sera accordé pour les questions de compréhension.**