

INF2610

# TP #1 : ProcessLab

## Groupe 3

École Polytechnique de Montréal

Automne 2022

Date limite de remise : Date de la séance du TP + 7 (avant 18h)  
**2 pts bonus** pour toute remise le jour ou le lendemain de votre séance de TP

### Présentation

Le *ProcessLab* a pour but de vous familiariser avec les appels système de la norme POSIX liés à la gestion de processus et threads (création, attente de fin d'exécution et terminaison).

### Prendre en main le lab

Prenez quelques instants pour vous familiariser avec la structure du répertoire sur lequel vous travaillerez durant ce TP. Vous trouverez dans le répertoire courant :

- `processlab.pdf` → L'énoncé du TP;
- `part1.c, part2.c` → Ce sont les fichiers de code que vous allez progressivement compléter pour répondre aux questions du TP;
- `output.txt` → Ces fichiers serviront à récupérer les résultats des traçages;
- `Makefile, libprocesslab` → Ce fichier et ce répertoire contiennent les commandes et les codes qui font fonctionner secrètement le TP...! Attention, vous ne devez pas les modifier.

**Il est conseillé de lire l'énoncé en entier avant de commencer le TP.** Il n'est pas demandé de traiter les erreurs éventuelles liées aux appels système. Par contre, si besoin est, à chaque fois que votre programme effectue un appel système (directement ou via une fonction de librairie), vous avez la possibilité d'afficher un message d'erreur explicite en cas d'échec de cet appel système. Pour ce faire, il vous suffit d'utiliser la fonction `perror` après l'appel système (ou l'appel de fonction de librairie). Consultez sa documentation!

## Instructions pour la partie 1

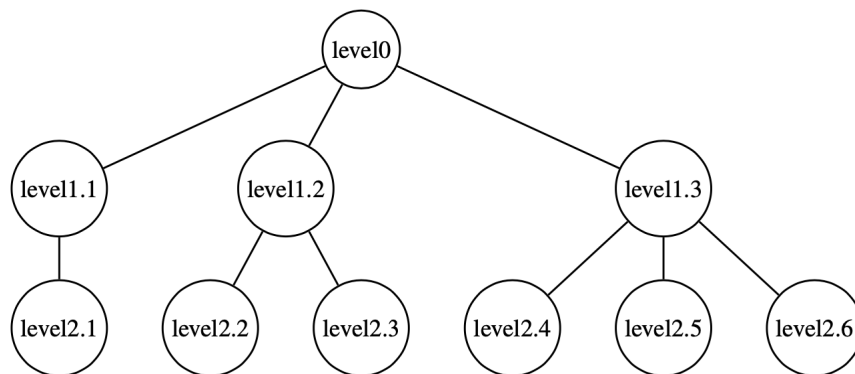




Figure 1: La hiérarchie des processus à recréer pour la partie 1.

### Question 1.1 (Création de l'arbre des processus) environ 40mn

Dans un premier temps, vous allez recréer l'arbre de processus décrit par la figure 1. **Le processus racine *level0* correspond au processus à partir duquel est exécutée la fonction *question1* du fichier *part2.c*.**

 Complétez la fonction *question1* du fichier *part1.c* afin de créer les processus selon la hiérarchie définie par la figure 1. A ce niveau, le traitement de chaque processus se limite à créer ses fils (s'il en a) et à attendre leurs terminaisons.

 Utilisez l'utilitaire *strace* avec les options *-fe trace=clone* pour vérifier si votre code recrée bien l'arbre des processus décrit par la figure 1. Ces options de *strace* permettent de limiter le traçage aux appels système liés à la création des processus. L'appel système *fork* (de la norme POSIX) se traduit sous Linux en un appel système *clone*. Ce dernier ne fait pas partie de la norme POSIX. La commande `./processlab 1` permet de tester le code de la fonction *question1*. Le programme *processlab.c* a un paramètre qui indique le numéro de la question à tester. Pour son traçage, utilisez la commande suivante : `strace -s 1000 -o output.txt -fe trace=clone ./processlab 1`.



**Attention :** Il n'est pas du tout exigé que votre code comporte des boucles *for*. Faites au plus simple!

L'ordre des processus décrit par la figure 1 importe! Par exemple, le processus *level1.1* doit être créé avant le processus *level1.2*, car il possède le même processus parent que *level1.2* mais il est situé plus à gauche dans la hiérarchie.

Un processus parent crée d'abord tous ses fils avant de se mettre en attente de leurs terminaisons.

Question 1.2 (Enregistrement des processus) ⌚ environ 15mn

✎ Complétez le code précédent afin que chaque processus (y compris le processus *level0*) fasse appel une fois et une seule fois à la fonction `registerProc` fournie dans le fichier `libprocesslab.h`. Cet appel doit être effectué en premier par chacun des processus. La fonction `registerProc` a besoin de quatre arguments :

1. le PID du processus appelant,
2. le PID du parent du processus appelant,
3. le niveau du processus appelant (exemples 2 dans le cas de *level2.3* et 0 pour *level0*), et enfin
4. le numéro du processus appelant à ce niveau-là (exemples 3 dans le cas de *level2.3* et 0 pour *level0*).

✎ Complétez maintenant le code afin que le processus *level0* fasse appel une fois et une seule fois à la fonction `printProcRegistrations` fournie dans le fichier `libprocesslab.h`. Cet appel doit être effectué juste après la fin de tous ses processus fils. Il permet d'afficher à l'écran les enregistrements réalisés par les processus (y compris le processus *level0*).



**Attention :** Avant l'enregistrement des processus, assurez-vous d'abord que l'arbre des processus de la figure 1 est bien reproduit par la question précédente, en consultant le fichier `output1.txt`. Pour chaque processus, assurez-vous aussi que vous passez les bons arguments à la fonction `registerProc`. Le passage de mauvais paramètres pourrait nuire au bon déroulement de votre programme.

Question 1.3 (Communication `_exit` - `wait`) ⌚ environ 15mn

✎ Complétez le code précédent afin que chaque processus parent (y compris le processus *level0*) puisse récupérer, via l'appel système `wait`, le nombre total de processus fils créés par ses descendants directs et indirects. Le processus *level0* doit afficher à l'écran le nombre total de fils créés par lui et tous ses descendants, juste avant l'appel à la fonction `printProcRegistrations`.

Question 1.4 (Transformation de processus) ⌚ environ 10mn

✎ Modifiez votre solution pour que le processus *level0* se transforme pour exécuter la commande suivante, juste après l'appel à la fonction `printProcRegistrations` : `ls -l`. Vous pouvez utiliser l'une des fonctions de la famille `exec`, mais soyez attentif à bien respecter la syntaxe de la fonction et la sémantique des arguments. En cas de doute, référez-vous aux *manpages* des fonctions concernées. La commande `whereis` permet de localiser certains exécutable (`whereis ls`).

## Instructions pour la partie 2

### Question 2.1 (Traitements séquentiels ou concurrents?)

🕒 environ 40mn

Cette partie vous propose de calculer, en mettant à contribution  $nb$  threads POSIX, la somme des  $m$  premiers nombres naturels strictement positifs :  $1 + 2 + \dots + m$ , où  $nb$  et  $m$  sont des constantes définies dans le fichier `part2.c`. Pour réaliser ce calcul, suivez les consignes suivantes :

- Les  $nb$  threads sont créés, l'un à la suite de l'autre, dans la fonction `question2` du fichier `part2.c`.
- Chaque thread créé exécute la fonction `contribution` du fichier `part2.c`. Cette fonction a un seul paramètre qui sert à récupérer le numéro d'ordre du thread qui l'exécute. Le numéro d'ordre est 0 pour le premier thread créé, 1 pour le second, et ainsi de suite.
- Si  $no$  est le numéro d'ordre du thread exécutant la fonction `contribution`, son traitement consiste à calculer la somme de tous les nombres entiers naturels figurant dans l'intervalle  $[(no * m/nb) + 1, (no + 1) * m/nb]$ . Le résultat de cette somme est récupéré dans `somme[no]`, où `somme` est un tableau, de  $nb$  entrées, défini dans le fichier `part2.c`.
- Après la création des deux threads, la fonction `question2` se contente, dans l'ordre, d'attendre la fin des threads créés et d'afficher à l'écran un message indiquant les valeurs des expressions `somme[0] + ... + somme[nb - 1]` et  $m * (m + 1) / 2$ . Ces deux valeurs devraient être égales.

🔧 Complétez les fonctions `question2` et `contribution` du fichier `part2.c` pour obtenir le comportement voulu. Une fois que vous aurez recompilé le TP, vous pourrez tester votre solution en exécutant la commande `./processlab 2`.

🔧 Utilisez l'utilitaire `time` pour récupérer les temps d'exécution en mode utilisateur, en mode noyau et réel de la commande `./processlab 2` :

`time ./processlab 2` pour  $nb = 1$ ,  $nb = 4$  et  $nb = 8$  (Rappel :  $nb$  est une constante définie dans le fichier `part2.c`).

Le temps d'exécution en mode utilisateur (resp. noyau) est le temps CPU consommé en mode utilisateur (resp. noyau). Le temps réel est le temps entre l'invocation et la fin de la commande. N'hésitez pas à consulter le manuel en ligne (`man time`) pour plus d'informations.

Remarquez que le temps réel pourrait être inférieur à la somme des deux autres temps, lorsque plusieurs threads sont mis à contribution. Aucune remise n'est demandée pour ce test de l'utilitaire `time`.

## Compilation, exécution et remise

Toutes vos solutions pour ce TP doivent être écrites dans les fichiers `part1.c` et `part2.c`. **Seuls ces deux fichiers et le fichier `output.txt` seront pris en compte pour évaluer votre travail; il est donc inutile, voire contre-productif, de modifier les autres fichiers que nous vous fournissons!**

Pour la partie 1, le fichier `output.txt` que vous allez soumettre est celui généré après avoir complété la question 1.1, 1.2, 1.3 ou 1.4.

### Compiler et exécuter le TP

Pour compiler le TP (initialement et après chacune de vos modifications), tapez la commande suivante (à partir du répertoire contenant le fichier `Makefile`) :

Console

```
$ make
```

Tapez `make mac` pour une compilation sans l'option `lrt`. Si la compilation se déroule sans erreur, vous pouvez ensuite exécuter le programme en tapant la commande :

```
Console
$ ./processlab n
```

Où `n` est 1 ou 2, dépendamment de la question à tester (`question1` ou `question2`).

## Soumettre votre travail

Votre travail doit être déposé sur le site moodle **avant la date limite indiquée plus haut**. Aucune remise au delà de la date limite ne sera acceptée.

Pour soumettre votre travail, créez d'abord l'archive de remise en effectuant :

```
Console
$ make handin
```

Cela va créer le fichier `handin.tar.gz` que vous devrez soumettre sur le site moodle après avoir ajouté au nom du fichier `handin` vos matricules. Attention, vérifiez bien que tous les fichiers nécessaires à l'évaluation de votre travail sont bien inclus dans le fichier soumis.

## Evaluation

Ce TP est noté sur 20 points répartis comme suit :

- /6.0 pts : Question 1.1
- /2.0 pts : Question 1.2
- /3.0 pts : Question 1.3
- /2.0 pts : Question 1.4
- /5.0 pts : Question 2.1
- /2.0 pts : Code (clarté et respect des consignes).