

TP #1: ProcessLab

École Polytechnique de Montréal

Hiver 2022 — Durée: 7 jours

Présentation

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

Vous aurez aussi l'occasion d'expérimenter l'outil de traçage *strace* de Linux. Ce dernier permet de récupérer tous les appels système effectués par un processus et éventuellement ses processus descendants. Il fournit pour chaque appel système, le PID du processus appelant, le nom de l'appel système, ses arguments et sa valeur de retour.

Il est conseillé de lire l'énoncé en entier avant de commencer le TP.

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;
- `output1.txt, output2.txt, output3.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.

Voici quelques points très importants à retenir:

- Tous les TPs du cours se dérouleront sur une plateforme de type Linux, avec des programmes écrits en C. Contrairement à ce que vous avez pu faire dans vos cours précédents, vous utiliserez de simples éditeurs de texte pour modifier les fichiers de code. La compilation se fera séparément, dans une fenêtre de terminal. N'oubliez pas que vous devez recompiler le TP à chaque fois que vous modifiez les fichiers de code, sinon vos modifications n'auront aucun effet! Vous trouverez plus de détails dans la section *Compilation, exécution et remise* de ce document.
- Il n'est pas demandé que votre solution finale traite 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 bibliothèque), vous avez la possibilité d'imprimer 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 bibliothèque en question. Consultez sa documentation!

Instructions pour la partie 1


Question 1.1 (*printf* ou *write*?) environ 20mn


Par défaut, la fonction `printf` de la librairie `stdio.h` du langage *C* possède un tampon dans lequel elle stocke temporairement les messages à afficher sur la sortie standard. Elle affiche, via un appel système `write`, le contenu du buffer, dès qu'une ligne (indiquée par un caractère de fin de ligne `\n`) est constituée ou encore le buffer est plein. Cela permet de faire l'économie d'appels système trop fréquents quand ce n'est pas nécessaire. Nous allons exploiter cette caractéristique pour afficher le message suivant:

```
75dbcb01f571f1c32e196c3a7d27f62e (printed using write)
75dbcb01f571f1c32e196c3a7d27f62e (printed using printf)
```

Les consignes à respecter sont les suivantes :


- La première ligne du message "75dbcb01f571f1c32e196c3a7d27f62e (printed using write)\n" doit être affichée en utilisant un seul appel système `write`.
- La deuxième ligne "75dbcb01f571f1c32e196c3a7d27f62e (printed using printf)" doit être affichée en utilisant la fonction `printf` pour chaque caractère du message.
- Dans votre code *C*, vous devez commencer par les appels `printf` **avant** l'appel système `write`. N'utilisez pas la fonction `fflush`. Vous devez obtenir le comportement voulu en utilisant le caractère de fin de ligne (`\n`) seulement.

 Complétez la fonction `question1` du fichier `part1.c` pour obtenir le comportement voulu. Une fois que vous aurez recompilé le TP, vous pourrez tester votre solution en exécutant `./processlab 1` dans un terminal. Le programme `processlab.c` a un paramètre qui indique le numéro de la question à tester.


 Utilisez l'utilitaire `strace` avec les options suivantes pour récupérer, dans le fichier `output1.txt`, les appels système `write` effectués par `./processlab 1`:


```
strace -s 100 -o output1.txt -e trace=write ./processlab 1.
```

Question 1.2 (*printf* sans *bufférisation*?) environ 20mn

 Complétez la fonction `question1` du fichier `part1.c` en dupliquant le code que vous avez ajouté pour afficher le message précédent. Mais cette fois vous allez précéder ce traitement "cloné" par un appel à la fonction `setvbuf` de la librairie `stdio.h`. Cet appel doit rendre "non bufferisée" la sortie standard (`stdout`) de l'appelant. Consultez la documentation de la fonction `setvbuf` (`man setvbuf`):

```
int setvbuf(FILE *stream, char buffer, int mode, size_t size).
```

 Utilisez le paramètre de la fonction `question1` pour qu'elle réalise le premier affichage (de la question 1.1), si elle est appelée avec comme argument la valeur 1. Elle réalise le second, si elle est appelée avec comme argument la valeur 2.

 Utilisez l'utilitaire `strace` avec les options suivantes pour récupérer, dans le fichier `output2.txt`, les appels système `write` effectués à partir de `question1(2)`:

```
strace -s 100 -o output2.txt -e trace=write ./processlab 2.
```

Remarquez les nombres d'appels système `write` effectués par les appels `question1(1)` et `question1(2)`.

Instructions pour la partie 2

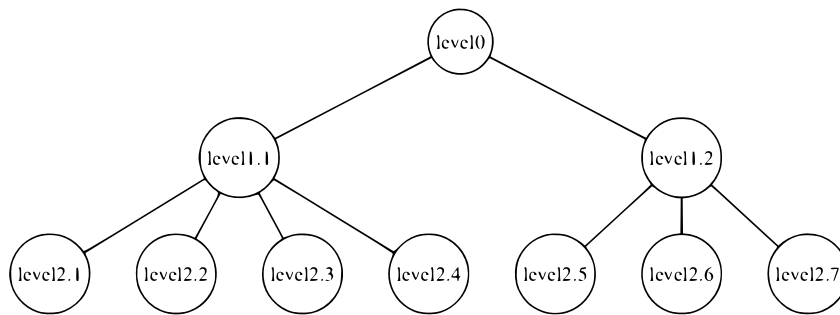


Figure 1: La hiérarchie des processus à recréer pour la partie 2.

Question 2.1 (Création de l'arbre des processus) ⌚ environ 35mn

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 *question2* du fichier *part2.c*.**

📖 Complétez la fonction `question2` du fichier `part2.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 3` permet de tester le code de la fonction `question2`. Pour son traçage, utilisez la commande suivante : `strace -s 100 -o output3.txt -fe trace=clone ./processlab 3`.



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



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 `output3.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.

📖 Complétez le code précédent afin que le processus *level0* fasse appel une fois et une seule fois à la fonction `printProcRegistrations` qui vous est 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*).

Question 2.3 (Communication via `_exit` - `wait`) ⌚ environ 20mn

📖 Complétez le code précédent afin que chaque processus (y compris le processus *level0*) puisse calculer le nombre total de fils créés par lui-même et ses descendants. Le nombre calculé par chaque processus (hormis le processus *level0*) est communiqué, via un appel à la fonction `_exit`, à son processus parent. Le processus *level0* va se contenter d'afficher à l'écran le nombre total de fils créés par lui et ses descendants, juste avant l'appel à la fonction `printProcRegistrations`.

Question 2.4 (Transformation de processus) ⌚ environ 20mn

📖 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`).

Compilation, exécution et remise

Toutes vos solutions pour ce TP doivent être écrites dans les fichiers `part1.c` (pour les questions 1.1 et 1.2) et `part2.c` (pour les questions 2.1, 2.2, 2.3 et 2.4). **Seuls ces deux fichiers et les fichiers `output1.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 2, le fichier `output3.txt` que vous allez soumettre peut être celui généré avant ou après avoir complété les questions 2.2, 2.3 et 2.4.

Compiler et exécuter le TP

Pour compiler le TP initialement et après chacune de vos modifications, tapez la commande (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, 2 ou 3 (dépendamment de la question `question1(1)`, `question1(2)` ou `question2()` à tester).

Soumettre votre travail

Votre travail doit être déposé sur le site moodle **avant la date indiquée sur le site moodle**. Aucune autre forme de remise de votre travail n'est acceptée. Les retards ou oublis sont sanctionnés comme indiqué sur la page Moodle du cours.

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:

- /2.5 pts: Question 1.1
- /2.0 pts: Question 1.2
- /5.5 pts: Question 2.1
- /2.0 pts: Question 2.2
- /2.0 pts: Question 2.3
- /2.0 pts: Question 2.4
- /4 pts: Code (clarté et respect des consignes).