INF2610

TP #1 : Appels système, processus et threads Groupe 01L

Polytechnique Montréal

Automne 2024 Date de remise: Voir le site Moodle du cours Pondération: 10%

Présentation

Ce travail pratique a pour but de vous familiariser avec l'environnement de programmation des laboratoires, les appels système de la norme POSIX liés à la gestion de fichiers et à la gestion des processus et threads (création, attente de fin d'exécution et terminaison).

Prise en charge du laboratoire

Prenez quelques instants pour vous familiariser avec la structure du répertoire sur lequel vous travaillerez durant ce TP. Vous trouverez dans le répertoire courant trois dossiers représentant chacun une section du TP. Chaque section vous sera présentée plus bas.

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!

Section 1 : Environnement de programmation et appels système en assembleur

Vous trouverez dans le répertoire:

- systemcalls_part1.c, systemcalls_part2.c

 Ce sont les fichiers de code que vous allez progressivement compléter pour répondre aux questions de cette section;
- systemcalls_output1.txt, systemcalls_output2.txt → Ces fichiers serviront à récupérer les résultats d'exécution et de traçage;
- Makefile → Ce fichier contient les commandes de compilation. Attention, vous ne devez pas le modifier.

Appels système liés à la gestion de fichiers

Question 1.1 (Appels système open, read, write et close) environ 25mn

Complétez le fichier systemcalls_part1.c pour qu'il réalise, en utilisant les appels système open, read, write et close, le traitement suivant :

- ouvrir le fichier nommé systemcalls_output2.txt en mode écriture et l'option O_TRUNC;
- afficher à l'écran le texte "Saisissez votre texte suivi de CTRL-D :\n". La combinaison des touches CTRL et D (^D) indique une fin de fichier;
- lire caractère par caractère des données à partir du clavier jusqu'à la rencontre d'une fin de fichier (^D);
- sauvegarder dans le fichier systemcalls_output2.txt chaque caractère lu à partir du clavier;
- fermer le fichier systemcalls_output2.txt.

N'oubliez pas d'ajouter les directives d'inclusion. Consultez le manuel en ligne pour les directives d'inclusion requises ainsi que les signatures des fonctions utilisées. Assurez-vous que l'appel système open ne retourne pas d'erreur en testant la valeur de retour. En cas d'erreur, vous devez afficher sur la sortie erreur un message du type "Appel système open a échoué" puis terminer le programme.

Pour compiler le programme, il suffit de lancer la commande :

gcc systemcalls_part1.c -o systemcalls_part1 ou encore make -B systemcalls_part1 La commande ./systemcalls_part1 permet de tester l'exécutable systemcalls_part1

Par défaut, la fonction fprintf de la librairie stdio.h du langage C possède un tampon (buffer) dans lequel elle stocke temporairement les messages à écrire 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:

77dbcb01f571f1c32p196c3a7d27f62e (printed using write) 77dbcb01f571f1c32p196c3a7d27f62e (printed using fprintf)

I 2.2.1. Complétez la fonction part21 du fichier systemcalls_part2.c pour obtenir le comportement voulu. Cette fonction est appelée lorsque le paramètre du programme systemcalls_part2.c est 1. Compilez le programme pour générer le code executable dans un fichier nommé systemcalls_part2.

```
Question 1.3 (fprintf sans bufférisation?) • environ 20mn
```

 $$\mathbb{R} = 2.3.1.$$ Complétez la fonction part22 du fichier systemcalls_part2.c pour qu'elle débute par un appel à la fonction setvbuf, de la librairie stdio.h, avant de réaliser un traitement identique à celui de la fonction part21. L'appel à la fonction setvbuf 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)
```

La fonction part22 est appelée, si l'exécutable systemcalls_part2 est lancé avec comme argument la valeur 2.

Compilation et exécution

Pour compiler les fichiers systemcalls_part1.c et systemcalls_part2.c, vous pouvez utiliser le fichier Makefile fourni. Les commandes suivantes créent les fichiers exécutables correspondants :

```
Console

make -B systemcalls_part1

make -B systemcalls_part2
```

Lancez la commande make pour une compilation de tous les fichiers. Si la compilation se déroule sans erreur, vous pouvez ensuite exécuter l'un des programmes en saisissant ./ suivi du nom de l'exécutable du programme. Toutes vos solutions pour cette section doivent être écrites dans les fichiers systemcalls_part1.c, systemcalls_part2.c et systemcalls_output1.txt, systemcalls_output2.txt.

Section 2: Processus et Threads

Vous trouverez dans le répertoire:

- processes_part1.c, processes_part2.c | → Ce sont les fichiers de code que vous allez progressivement compléter pour répondre aux questions de cette section;
- processes_output.txt → Ce fichier sert à récupérer les résultats d'exécution et de tracage;
- Makefile → Ce fichier contient les commandes de compilation. Attention, vous ne devez pas le modifier.

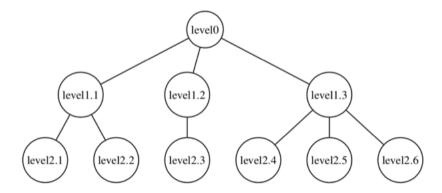


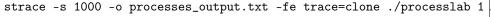
Figure 1: La hiérarchie des processus à recréer pour la partie 1.

Question 2.1 (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 processes_part1.c.

[Complétez la fonction question1 du fichier processes_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 enfants (s'il en a) et à attendre leurs terminaisons.

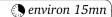
I'm Utilisez l'utilitaire strace avec les options -fe trace=clone pour vérifier si votre code recrée bien l'arbre des processus décrit par la figure 1. Ces options de strace permettent de limiter le traçage aux appels système liés à la création des processus. L'appel système fork (de la norme POSIX) 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 processes_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 level 1.2, car il possède le même processus parent que level 1.2 mais il est situé plus à gauche dans la hiérarchie.

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

Question 2.2 (Enregistrement des processus)



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 enfants. 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 processes_output.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 2.3 (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 enfants créés par ses descendants directs et indirects. Le processus *level0* doit afficher à l'écran le nombre total de enfants créés par lui et tous ses descendants, juste avant l'appel à la fonction printProcRegistrations.

Question 2.4 (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 : $\boxed{ 1s - 1}$.

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 exécutables (whereis ls).

Instructions pour la partie 2

Question 3.1 (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 : $1+2+\ldots +m$, où nb et m sont des constantes définies dans le fichier processes_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 processes_part2.c.
- Chaque thread créé exécute la fonction contribution du fichier processes_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 processes_part2.c.
- Après la création des nb 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 $somme[0]+\ldots+somme[nb-1]$ et m*(m+1)/2. Ces deux valeurs devraient être égales.

**Complétez les fonctions question2 et contribution du fichier processes_part2.c pour obtenir le comportement voulu. Une fois que vous aurez recompilé cette partie, 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 processes_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 et exécution

Toutes vos solutions pour cette section doivent être écrites dans les fichiers processes_part1.c et processes_part2.c. Seuls ces deux fichiers et le fichier processes_output.txt seront pris en compte pour évaluer votre travail. Pour la partie 1, le fichier processes_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 cette section

Pour compiler cette section (initialement et après chacune de vos modifications), tapez la commande suivante (dans le répertoire de la section 2) :



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



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

Section 3 : Défis

Dans cette dernière partie du TP, on vous demande de résoudre deux exercices en vous basant sur les notions que vous avez apprises dans les sections 1 et 2. Vous trouverez dans le répertoire:

- challenges_part1.c, challenges_part2.c, challenges_part1.h, challenges_part2.h → Ce sont les fichiers de code que vous allez progressivement compléter pour répondre aux questions de cette section;
- generate_directory → Ce fichier servira à générer une arborescence de répertoires (plus de détails seront fournis plus bas);
- test_part2.o
 Ce fichier servira à tester votre solution de la partie 2;
- Makefile → Ce fichier contient les commandes de compilation et de remise. Attention, vous ne devez pas le modifier.

Instructions pour la partie 1

```
Question 4.1 (Exploration d'un arbre)
```

Dans cette partie, on vous demande de nous aider à explorer un arbre de processus. Utilisez l'exécutable generate_directory pour générer ce dernier en lui fournissant vos matricules étudiants. Voici un exemple: ./generate_directory 1234567 7654321

Vous devriez maintenant apercevoir un répertoire root. Ce dernier est composé d'un nombre arbitraire de sous-répertoires et de fichiers texte. Votre mandat est le suivant:

- Vous devez trouver le nombre de fichiers texte présents dans l'ensemble de l'arborescence en utilisant des processus. À chaque nouveau sous-répertoire rencontré, un nouveau processus doit être créé. Une fois le nombre total de fichiers texte calculé, vous devez l'afficher dans le terminal.
- Nous souhaitons également avoir davantage d'information sur l'arborescence des répertoires. On vous demande de générer le fichier texte challenges_output.txt. Dans ce dernier, nous souhaitons récupérer les informations suivantes pour **chaque répertoire rencontré**:
 - L'emplacement du répertoire;
 - Le numéro d'identification du répertoire (Indice: on se souvient qu'à chaque répertoire rencontré, un nouveau processus est créé);
 - Le numéro d'identification du répertoire supérieur immédiat;
 - La liste des fichiers texte se trouvant dans le répertoire.

Attention! Le contenu du fichier challenges_output.txt doit être compréhensible. Les informations sur un répertoire donné doivent être regroupées ensemble. Toutefois, vous n'avez pas à vous soucier de l'ordre dans lequel l'écriture est réalisé (un répertoire d'un niveau inférieur pourrait apparaître avant celui d'un niveau supérieur et vice-versa). Voici à quoi devrait ressembler le fichier challenges_output.txt:

Complétez la fonction main du fichier challenges_part1.c pour obtenir le comportement voulu. Vous pouvez compiler cette partie avec la commande make. Lancez ensuite l'exécution avec la commande ./challenges_part1.

Instructions pour la partie 2

Question 5.1 (Calcul matriciel)

Dans cette partie, on vous demande d'effectuer un produit matriciel. Vous devez implémenter la fonction multiply du fichier challenges_part2.c. Voici quelques informations supplémentaires:

- Vous ne pouvez pas changer la signature de la fonction;
- La fonction doit retourner une matrice C résultat du produit matriciel A × B (A et B étant passé en paramètre);
- Lorsque que le produit matriciel est invalide, la fonction doit retourner NULL.

Complétez la fonction multiply du fichier challenges_part2.c pour obtenir le comportement voulu. Durant votre implémentation, vous pouvez compiler cette partie avec la commande:

gcc -00 -std=gnu11 -Werror -Wall -Wno-unused-result -Wno-unused-variable
-o challenges_part2 ./challenges_part2.c -lrt.

Vous devez d'abord décommenter la fonction main se trouvant dans le ficher. Lancez ensuite l'exécution avec la commande ./challenges_part2.

Une fois confiant(e)s de votre implémentation, vous pouvez tester celle-ci avec la commande make test (n'oubliez pas de recommenter la fonction main d'abord). Lancez ensuite l'exécution des tests avec la commande ./test_part2.

Pour recevoir l'entièreté des points pour ce numéro, une condition nécessaire, mais non suffisante est de passer tous les tests sur les ordinateurs du laboratoire.

Pour soumettre votre travail, créez d'abord l'archive de remise en lançant la commande dans le répertoire de la section 3:

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.

Évaluation

Ce TP est noté sur 20 points répartis comme suit :

/5.0 pts : Section 1/5.0 pts : Section 2/8.0 pts : Section 3

• /2.0 pts : Code (clarté et respect des consignes).