INF2610

TP #0: InitLab

École Polytechnique de Montréal

Hiver 2023
Date de remise: Voir le site Moodle du cours
Pondération : 5%

Attention :
- Tous les laboratoires du cours auront lieu dans les laboratoires `Linux` du département `GIGL` et dans un environnement de programmation basé sur le langage `C`. [Pour les intéressés, les codes sources de Linux sont disponibles et accessibles à tous à l'adresse : kernel.org].
- Tous les travaux pratiques sont à réaliser en équipe de deux d'une même section de laboratoire.
- Les laboratoires complètent le cours théorique et peuvent faire l'objet de questions d'examen.

Présentation

Le laboratoire *InitLab* a pour but de vous familiariser avec l'environnement de programmation des laboratoires et les appels système de la norme POSIX liés à la gestion de fichiers.
Vous aurez également à expérimenter des appels système en assembleur ainsi que l'outil de traçage *strace*. Ce dernier permet de récupérer tous les appels système effectués par un processus. Il fournit pour chaque appel système, le numéro du processus appelant, le nom de l'appel système, ses arguments et sa valeur de retour.

Prenez quelques instants pour vous familiariser avec la structure du répertoire sur lequel vous travaillerez durant ce TP. Vous trouverez dans le répertoire du TP :
- `initlab.pdf` ➔ L'énoncé du TP ;
- `minisys.asm`, `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`, `output4.txt` ➔ Ces fichiers serviront à récupérer les résultats d'exécution et de traçage ;
- `Makefile` ➔ Ce fichier contient les commandes de compilation et de génération de l'archive de remise. Attention, vous ne devez pas le modifier.

Environnement de programmation & appels système en assemblleur

Question 1.1 *(Édition, Compilation et exécution des programmes)* *environ 30mn*

Vous utiliserez, via un terminal, de simples commandes pour éditer et compiler des programmes. La commande `gcc` (GNU Compiler Collection) sera utilisée pour la compilation de programmes écrits en langage `C` et la génération de leurs fichiers exécutables. N'oubliez pas de **recompiler** un programme à chaque fois que vous le modifiez, sinon vos modifications n'auront aucun effet !
Pour lancer l'exécution d'un programme, il suffit de saisir le nom de son fichier exécutable, si celui-ci est dans l'un des répertoires enregistrés dans la variable d'environnement `PATH`. Si ce n'est pas le cas, vous devez spécifier le chemin absolu ou relatif du fichier. Une fois le fichier exécutable lancé, il faut attendre la fin de son exécution avant de pouvoir lancer une nouvelle commande. Cependant, il est possible de forcer sa terminaison en appuyant simultanément sur les touches `CTRL` et `C` (`^C`).

☞ 1.1.1. Expérimentez, dans l'ordre, les commandes suivantes en prenant le soin d'enregistrer, dans le fichier `output1.txt`, les commandes exécutées ainsi que les résultats obtenus (pour remplir le fichier, vous pouvez utiliser la commande `echo` et les opérateurs de redirection $>$ et $>>$) :

Commande	**Description**
`pwd`	affiche le chemin complet du répertoire courant
`cd [chemin]`	change de répertoire (`[chemin]` doit être remplacé par le chemin qui mène vers le répertoire du TP).
`ls`	affiche le contenu du répertoire courant
`ls | sort`	affiche le contenu du répertoire courant trié
`echo "message1"`	affiche le texte `"message1"` à l'écran

`echo "message1" > foo.txt ;`
`echo "message2" >> foo.txt;`
`cat foo.txt`
	crée le fichier `foo.txt`, enregistre le texte `"message1"` dans le fichier, ajoute le texte `"message2"` à la fin du fichier, puis affiche le contenu du fichier

`unlink foo.txt ; ls`	supprime le fichier `foo.txt`
`uname`	affiche certaines informations concernant le système
`uname -s`	affiche le nom du noyau
`uname -r`	affiche la version du noyau
`uname -o`	affiche le nom de la combinaison noyau et interface utilisateur
`uname -m`	affiche le nom de l'architecture matérielle de la machine

☞ 1.1.2. Exécutez les commandes appropriées pour réaliser chacun des traitements suivants :

a. Afficher les chemins contenus dans la variable `PATH` avec la commande : `echo $PATH`.
b. Déterminer dans quel répertoire se trouve le binaire `sleep` grâce à la commande `which`.
c. Dormir pendant au plus 1 seconde (`sleep 1`) avec :
   — un chemin relatif au répertoire dans lequel se trouve l'exécutable `sleep` ;
   — son chemin absolu ;
   — un chemin relatif au répertoire courant.
d. Dormir pendant au plus 1000 secondes (`sleep 1000`) et interrompre ensuite la commande (un arrêt forcé).

Ajoutez, au fichier `output1.txt`, les commandes exécutées ainsi que les résultats obtenus.

Question 1.2 *(Appels système en assembleur)* *environ 25mn*

À partir d'un programme `C`, les services du noyau d'un système d'exploitation sont accessibles via des fonctions (appels système). Ces fonctions constituent donc une API (Application Programming Interface) entre les programmes `C` et le noyau.
Par exemple, la fonction `write` de cette API permet de solliciter le service d'écriture dans un fichier. C'est cette fonction qui va se charger de générer l'interruption logicielle des appels système et de communiquer au noyau, via des registres, le numéro de l'appel système ainsi que les arguments de cet appel. Le rôle de cette interruption est de basculer l'exécution en mode noyau et de lancer l'exécution du code noyau associé à l'appel système. Le nom de l'interruption et les registres utilisés, pour le transfert, dépendent de l'architecture matérielle de la machine hôte du système.
Par exemple, pour l'architecture `x86_64`, un appel système est supposé avoir au plus 6 arguments et les registres utilisés pour le transfert sont :

- `rax`, pour le numero de l'appel système,
- `rdi, rsi, rdx, r10, r8` et `r9`, pour les 6 paramètres (`rdi` pour le 1^{er} paramètre, ..., et `r9` pour le $6^{\text{ième}}$ paramètre).

Le résultat de l'appel système est récupéré dans le registre `rax`.

Il est également possible, à partir d'un programme en assembleur, de provoquer directement, via une instruction assembleur (`syscall`, `INT`, etc.), l'interruption logicielle qui va faire basculer l'exécution en mode noyau et exécuter le traitement associé à un appel système. Par exemple, pour l'architecture `x86_64`, le programme assembleur `minisys.asm` invoque le noyau pour afficher un message sur la sortie standard. Un extrait du code source est reproduit ici :

```
mov     rdi, stdout       ; argument 1 : descripteur de fichier
mov     rsi, msg          ; argument 2 : pointeur vers la chaîne
mov     rdx, msg_len      ; argument 3 : longueur de la chaîne
mov     rax, sys_write    ; numéro de l'appel système à invoquer
syscall                   ; basculement en mode noyau et
                          ; lancement du code associé à sys_write
```

Vous trouverez, si besoin est, des informations supplémentaires sur les appels système en assembleur sous `Linux` et la liste des appels système aux adresses suivantes :
`https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux`
`https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl`

☞ 1.2.1. Compilez et testez le code `minisys.asm` en lançant successivement la commande :
`nasm -f elf64 minisys.asm && ld -s -o minisys minisys.o && ./minisys`
Pour la génération de l'exécutable `minisys`, vous pouvez également utiliser le `Makefile` fourni :
`make -B minisys`

☞ 1.2.2. Affichez la valeur de retour du code tout de suite après son exécution avec la commande :
`echo $?`

☞ 1.2.3. Remplacez la valeur passée à l'appel système `sys_exit` par 10. Testez-le et affichez la valeur de retour du code.

☞ 1.2.4. Complétez le code `minisys.asm` pour faire une pause de 3 secondes et 100 nanosecondes dans le programme, en ajoutant un appel système `sys_nanosleep` (de numéro 35) à l'endroit indiqué par un commentaire dans le fichier `minisys.asm`. Testez-le.

☞ 1.2.5. Ajoutez un appel système `sys_write`, juste après l'appel système `sys_nanosleep`, pour afficher à l'écran le message `"Fin de la pause!\n"`. Testez-le.

☞ 1.2.6. Commentez l'appel à `sys_exit` puis testez le code. **Rétablissez** ensuite l'appel système. Remarquez la nécessité de cet appel système.

Appels système liés à la gestion de fichiers

Question 2.1 *(Appels système open, read, write et close)* *environ 25mn*

☞Complétez le fichier `part1.c` pour qu'il réalise, en utilisant les appels système `open, read, write` et `close`, le traitement suivant :

- ouvrir le fichier nommé `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 `output2.txt` chaque caractère lu à partir du clavier ;
- fermer le fichier `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 part1.c -o part1` ou encore `make -B part1`
La commande `./part1` permet de tester l'exécutable `part1`

Question 2.2 *(`fprintf` ou `write`?)* *environ 30mn*

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 :

```
77dbcb01f571f1c32e196c3a7d27f62e (printed using write)
77dbcb01f571f1c32e196c3a7d27f62e (printed using fprintf)
```

Les consignes à respecter sont les suivantes :

- La première ligne du message "77dbcb01f571f1c32e196c3a7d27f62e (printed using write)\n" doit être affichée en utilisant un seul appel système `write`.
- La deuxième ligne "77dbcb01f571f1c32e196c3a7d27f62e (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.

☞2.2.1. Complétez la fonction `part21` du fichier `part2.c` pour obtenir le comportement voulu. Cette fonction est appelée lorsque le paramètre du programme `part2.c` est 1. Compilez le programme pour générer le code executable dans un fichier nommé `part2`.

☞2.2.2. Utilisez la commande suivante pour récupérer, dans le fichier `output3.txt`, les appels système `write` effectués par `./part2 1` :
`strace -s 1000 -o output3.txt -e trace=write ./part2 1`.

Question 2.3 *(fprintf sans bufférisation ?)* *environ 20mn*

☞ 2.3.1. Complétez la fonction `part22` du fichier `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 `part2` est lancé avec comme argument la valeur 2.

☞ 2.3.2. Utilisez la commande suivante pour récupérer, dans le fichier `output4.txt`, les appels système `write` effectués par `./part2 2` :
`strace -s 1000 -o output4.txt -e trace=write ./part2 2`.
Remarquez les nombres d'appels système `write` effectués par les deux versions d'affichage.

Compilation, exécution et Remise

Pour compiler les fichiers `minisys.asm`, `part1.c` et `part2.c`, vous pouvez utiliser le fichier `Makefile` fourni. Les commandes suivantes créent les fichiers exécutables correspondants :

Console
```
make -B minisys
make -B part1
make -B 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 ce TP doivent être écrites dans les fichiers `minisys.asm`, `part1.c`, `part2.c` et `outputi.txt` pour $1 = 1$ à 4. Seules ces fichiers seront pris en compte pour évaluer votre travail ; il est donc inutile, voire contre-productif, de modifier ou d'ajouter d'autres fichiers.
Votre travail doit être déposé sur le site moodle **avant la date indiquée sur le site Moodle du cours**. Aucune autre forme de remise de votre travail n'est acceptée. Les retards ou oublis sont sanctionnés (voir le site Moodle pour les pénalités de retards).
Pour soumettre votre travail, créez d'abord l'archive de remise en lançant la commande :

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 :
— /3.0 pts : Question 1.1
— /3.5 pts : Question 1.2
— /4.5 pts : Question 2.1
— /3.5 pts : Question 2.2
— /3.5 pts : Question 2.3
— /2.0 pts : Code (clarté et respect des consignes).