Ecole Ingénieurs 2000 - Université de Marne-la-Vallée
Filière Informatique et Réseaux
2ème année

Projet de Système

******

vIRtuel machin

******

Shell permettant d'exécuter des commandes à la façon de la programmation objet mais sans appel bloquant.

Compte rendu de : **Tom MIETTE,**
Sébastien MOURET

Fait le : **29 février 2008**

Sommaire

Introduction

Ce logiciel est constitué de trois exécutables, **launch**, **oreader** et **vIRm**. Ce dernier, permet de lancer un shell qui attend certains types de commandes. Par exemple, nous pouvons créer un objet grâce à la commande '**new**' suivie du nom de l'objet et le nom de la librairie '**.so**' qui doit représenter notre objet. Cette librairie sera exécuter par le shell dans un processus à part à l'aide du programme launch. Le dernier exécutable '**oreader**' permet de lire ce que la librairie voudra envoyer sur sa sortie standard.

I. Détails des applications

1. Le shell : vIRm

Le shell est responsable de gérer et d'interpréter les commandes que l'utilisateur entre dans le terminal. Pour cela, il dispose de plusieurs fonctions de parsage présentes dans le fichier parse.c. Le parsage de la ligne de commande entrée par l'utilisateur est assez permissive (c'est un buffer de taille fixe), il est donc normal que le shell n'interprète pas les commandes si elles sont entrées de façon erronées. Lors du parsing de la cette ligne de commande, les fonctions de parsing remplissent une structure appelée 'command', cette dernière est utilisé par le shell pour exécuter les actions spécifiques.

Les actions possibles et compréhensibles du shell sont :

- **new 'nom_de_l'objet(nom_de_la_librairie)'**: pour la création de nouveaux objets. La fonction permettant de gérer cette création d'objet est chargée de créer un nouveau processus et de stocker son nom ainsi que son pid ainsi qu'un tube permettant de communiquer avec lui. Ces paramètres sont stockes dans un tableau de taille fixe de structures d'objets. Le processus fils sera recouvert par l'ouverture de la librairie par la fonction **execl**.

- **free 'nom_de_la_variable'** : cette création permet de libérer une variable représentant une zone de la mémoire partagée utilisée par les processus. Lors de l'appel à la fonction **work** d'un objet celui-ci écrit des données en mémoire partagée, ces données peuvent être associées à une variable afin d'être réutilisées par un autre objet. Ces informations sont stockées dans un certain bloc de 1024 octets en mémoire. Nous associons ce bloc a la variable entrée par l'utilisateur grâce a une structure nommée Bloc. Ces structures sont stockées dans un tableau de taille fixe.

- **work** : appel à la fonction **work** présente dans la librairie représentant l'objet créé. Les arguments de la commande peuvent être des entiers ou des chaines de caractères, ces données sont stockées dans un bloc mémoire et le numéro de celui ci est envoyé a l'objet (par l'intermédiaire du tuyau crée par le shell, la sortie de celui-ci étant redirige sur l'entrée standard du processus représentant l'objet).

- **waitall** : elle permet de notifier à tous les objets, que le shell attend qu'ils aient fini leurs taches. Le shell peut envoyer cette commande demandant l'attention des processus fils. A la réception de cette commande dans le shell, celui ci leur envoie sur l'entrée standard la commande **waitall** et les enregistre dans un tableau en tant que processus en attente. Lorsque les processus auront reçu cette fonction, ils enverront un signal au shell qui, a son tour les enregistrera comme 'prêt' ainsi, une fois tous les processus enregistres seront a l'état 'prêt' il s'occupera de leur envoyer un signal pour les redémarrer.

- **quit** : fonction permettant de quitter le shell. Cette fonction peut être appelée dans le shell pour lui signifier que l'utilisateur veut quitter l'application. Le shell envoi un signal SIGINT à ses fils pour les arrêter a leur tour. Un ctrl+c permet également de quitter le shell.

Le shell est responsable de gérer une mémoire partagée sous la forme d'un fichier mappé en mémoire et découpé sous forme de blocs de 1024 octets. Chaque bloc peut être utilise par les objets pour écrire ou lire des données.

Il est responsable de traiter les signaux envoyé par ses fils. Ces signaux permettent au shell d'exécuter une action particulière.

De plus, le shell doit ouvrir le tube nommé qui sera utilisé par le **oreader** afin que toutes les écritures sur **stdin** soient redirigées par un **dup** dans ce fichier.

2. Le lanceur : launch

Le launch est responsable d'ouvrir la librairie que le shell lui a fournie lors de la requête de création de l'objet par l'utilisateur via le **execl**. Cette librairie doit avoir une fonction '**void work(void* input, void* output)**' pour fonctionner.

Il doit récupérer le nom du fichier mappé par le shell via la ligne de commande (**execl**) et l'ouvrir pour le mapper à son tour (lecture + écriture). Ce fichier permettra a la fonction '**work**' de la librairie qu'il a ouvert d'écrire dans le bloc mémoire défini.

Il doit de plus initialiser son gestionnaire de signal pour être capable de se mettre en attente d'un signal SIGUSR2 provenant du shell. Nous verrons par la suite pourquoi.

Il est responsable de l'ouverture du tube nommé pour pouvoir effectuer des affichages sur sa sortie standard.

3. Le lecteur : oreader

Le **reader** n'est responsable que de l'ouverture du tube nommé utilisé par le shell et le lanceur pour afficher des éventuels **print** des objets sur la sortie standard de leur processus.

II. Communication des processus

1. Communication du shell avec ses fils (objets)

Le shell doit être capable de 'discuter' avec les objets qu'il crée grâce au lanceur. Pour cela nous avons mis en place un système de tube. Le shell écrit sur le tube (en entrée 1) et le processus redirige son entrée standard sur la sortie du même tube (0). Ainsi a chaque fois que le shell veut envoyer une tache à son fils (une tache est représentée par la structure Job), il écrit cette structure a l'entrée du tube. De ce fait, le processus fils n'a plus qu'à attendre grâce a un **read** de lire un objet Job afin de traiter les données (pid fonction **blocIN blocOUT**) provenant du shell.

Voici des exemples illustrant le fonctionnement de la communication du shell avec son fils.

Si l'utilisateur entre la commande suivante :

```
> monObjet.work(1,2,3,"bonjour")
```

Le shell après parsing de la commande va envoyer une structure Job contenant son pid, le nom de la fonction ici '**work**', le numéro du bloc de mémoire partagée ou '1,2,3,"bonjour"' a été écrit ainsi qu'un bloc libre pour d'éventuel écriture en mémoire par l'objet.

Du coté de l'objet celui ci lit son entrée standard sans discontinuer, des qu'il reçoit cette commande il n'a plus qu'à la traiter.

Avec un **waitfor** :

```
> monObjet.work(1,2,3,"bonjour") waitfor objetB
```

Ici le shell va commencer par mettre en pause le processus fils par un signal SIGSTOP et envoie une fonction **waitfor** a l'objet 'objetB' avec le pid de l'objet 'monObjet' afin qu'il le réveille avec un SIGCONT des qu'il aura fini ses propres taches (si le processus lit le **waitfor** c'est qu'il a fini de défiler son tube). Le shell peut ensuite envoyer la fonction **work** a l'objet 'monObjet'. L'inconvénient de cette méthode, c'est que l'objet 'monObjet' est bloqué avant d'arrivé sur la fonction **work** '**work(1,2,3,"bonjour")**' donc il peut être bloqué sur une autre tache (qu'il pourra finir uniquement après que l'objetB ait fini).

La commande **waitall** :

```
> waitall
```

Cette fonction est un peu particulaire. En effet, elle permet de mettre en 'PAUSE' tous les processus crées avant son appel. Pour cela, nous envoyons a tous nos fils la fonction **waitall** qui, quand ils la recevront (donc ils seront prêts) nous enverront un signal SIGUSR1. Des que nous aurons reçu des signaux de tous nos fils, nous pourrons les redémarrer grâce a un signal SIGUSR2.

La commande **free** :

```
> free x
```

Cette commande est toute simple, elle permet de libérer le bloc mémoire en question s'il existe.

Avec le **waitfor**

```
> free x waitfor notreObjet
```

Cette variante est un peu plus difficile à gérer. Pour la réaliser, nous envoyons (le shell pas nous) la fonction 'free' a notreObjet avec en paramètre le bloc d'entrée de la mémoire correspondant au bloc pointe par 'x'. L'objet est ensuite chargé de nous envoyer un signal SIGRTMIN+1 avec le numéro du bloc à libérer (numéro pointe par 'x').

2. Communication des objets avec le shell

Reprenons l'exemple ci-dessus pour comprendre ce que fait l'objet lorsqu'il reçoit une fonction **work**.

```
> monObjet.work(1,2,3,"bonjour")
```

A la réception de cette commande l'objet envoie le bloc mémoire d'entrée et le bloc sorti traduit à la fonction '**work**' de sa librairie. Traduit signifie qu'un numéro de bloc doit être au préalable multiplié à la taille d'un bloc (1024), et le pointeur sur le début de la zone du fichier mappé doit être incrémenté de cette valeur.

Une fois l'exécution de la fonction **work** terminée, le processus envoie deux signaux temps réels au shell. Ces deux signaux sont accompagné chacun d'un entier correspondant au bloc de mémoire d'entrée et de sortie. Ceci a pour effet de demander au shell de libérer ces zones. Celui ci s'exécutera si et seulement si ces zones ne sont pas réservées à des variables.

La commande **waitall** :

```
> waitall
```

A la réception de cette commande l'objet se mettent à attendre un signal SIGUSR2 provenant du shell pour nous signifier qu'il est temps de reprendre notre exécution. Cette attente se fait grâce a la fonction sigsuspend, cette fonction permet de gérer le fait de recevoir le signal attendu avant de se mettre en attente de ce même signal, sans quoi le processus serait suspendu a jamais !

III. Tests

Afin de tester le programme, nous avons mis en place quelques tests simples qui permettent de mettre en évidence le comportement du logiciel. Les tests se font avec les librairies externes (.so) qui se trouvent dans le répertoire "objects".

Voici une brève présentation de ce que savent faire ces librairies :

- **int2string.so** (présentée dans le sujet) lit un entier dans la zone mémoire en entrée (input) et écris cette entier sous forme de chaîne de caractères dans la zone mémoire de sortie (output)

- **printer.so** permet de lire la zone mémoire d'entrée sous forme de chaîne de caractères et de l'afficher sur la sortie standard du processus.

- **sleeper.so** récupère un entier dans la zone mémoire d'entrée et endort le processus courant pendant un temps égal à cet entier (en secondes). Cette libraire est très utile pour les tests d'attente entre objets.

- **calc.o** permet d'effectuer une opération simple (addition, soustraction multiplication ou division) entre deux entiers. Le processus lit deux entiers dans la zone mémoire d'entrée et un caractère définissant l'opération à effectuer (respectivement '+', '-', '*' ou '/'). Une fois l'opération effectuée, le processus écrit le résultat sur sa sortie standard et également dans la zone de mémoire de sortie.

Pour exécuter le programme, il faut vous placer dans le répertoire bin de l'application. Lancer d'une part le **vIRm** dans un terminal pour recevoir les commandes et le **oreader** dans un second terminal pour visualiser les affichages.

1. Test 1 : création d'objets et appels à work()

```
> new ob1(printer)

# créer un nouvel objet appelée "ob1" dont la méthode work() est liée à la librairie "printer.so". Note : new ob1("printer") fonctionne également

> ob1.work("First test...")

# appel la méthode work() de "printer.so"

> First test...

# affichage résultant

> new c(calc)

# créer un objet "c" lié à la libraire "calc.so"

> c.work(10,5,"+")

# demande un addition entre 10 et 5

    > 10 + 5 = 15; # affichage résultant

> quit

# quitte le shell
```

2. Test 2 : allocation, utilisation et libération de variables

```
> new c(calc)

> x=c.work(4,5,"*")
# stockage de la zone mémoire de sortie de work() dans "x"
```

```
      > 4 * 5 = 20; # affichage résultant

> y=c.work(x,x,"+")

# récupération de l'entier stocké dans "x" et sauvegarde du nouveau résultat dans
"y"

      > 20 + 20 = 40; # affichage résultant

> new is(int2string)
# créer un objet "is" convertisseur d'entier en string

> z=is.work(y)
# récupère la variable "y" et stocke la chaîne créée dans "z"

> new p(printer)
# créer un objet "p" afficheur

> p.work(z) # demande l'affichage de la variable "z"
    > 40 # affichage résultant

> free x # libère les variables "x", "y" et "z"
> free y
> free z

> quit
# quitte le shell
```

3. Test 3 : utilisation de waitfor avec work() et free

```
> new s(sleeper) # créer un objet "s" dormeur

> new p(printer) # créerun objet "p" afficheur

> new c(calc) # créer un objet "c" calculateur


> s.work(15) # endort l'objet "s" pendant 15 secondes

> p.work("Third test...") waitfor s
# demande un affichage à "p" une fois que "s" aura terminé toutes ces tâches

> x=c.work(25,11,"-") waitfor s
# demande un calcul à "c" une fois que "s" aura terminé toutes ces tâches et stocke
le résultat dans "x"

> s.work(10)
# poste une tâche dans "s" alors que celui-ci est occupé
# attente de 15 secondes environ ...

    > Third test... # affichage résultant une fois que "s" s'est réveillé
    > 25 - 11 = 14;

# "s" se rendort 10 secondes...

> s.work(x) # endort s pendant 14 secondes

> free x waitfor s
# demande la libération de "x" une fois le travail de "s" terminé

# attente de 14 secondes environ ...
# libération effective de x
```

```
> quit # quitte le shell
```

4. Test 4 : utilisation du waitall

```
> new s1(sleeper) # créer un objet "s1" dormeur

> new s2(sleeper) # créer un objet "s2" dormeur

> new p(printer) # créerun objet "p" afficheur


> s1.work(10) # endort s1 pendant 10 secondes

> s2.work(20) # endort s2 pendant 20 secondes

> waitall # attend la fin de tous les processus lances

> p.work("Fourth test...") # demande un affichage
#attente de tous les processus, soit 20 secondes environ ...

    > Fourth test... # affichage resultant


> quit # quitte le shell
```

IV. Bugs connus

1- Pas plus de 1024 octets de données dans une zone mémoire, nous ne prenons pas en charge le fait d'écrire dans un fichier dans le cas contraire.

2- Nous n'avons pas eu le temps de coder la fonction permettant de récupérer le signal SIGCHLD, lorsqu'un fils se termine de façon inopiné. Par exemple si vous créer un objet avec en argument le nom d'une librairie qui n'existe pas le processus sera crée et le shell fera comme si de rien n'était.

Conclusion

Ce projet a été l'occasion d'apprendre beaucoup sur la programmation c-systeme, ce qui nous a permis de comprendre en partie comment est géré un 'vrai' shell. Il nous a permis d'être à l'aise avec la manipulation des tubes ainsi que des processus.