HEIG-VD | REDS Institut Reconfigurable and Embedded Digital Systems

Systèmes d'exploitation (SYE)

Profs Daniel Rossier, Fiorenzo Gamba, Marina Zapater
Assistants : David Truan, Lucas Elisei, Mattia Gallacchi

Appels systèmes et processus

lab02 (04.10.2021)

Objectifs de laboratoire

Ce laboratoire se focalise sur la notion d'appel système au niveau de l'espace utilisateur et du noyau. Il permettra de se familiariser davantage avec le code du *shell* et de l'ensemble des fichiers qui composent SO3. De plus, le laboratoire consacre une étape sur la création de processus et le chargement d'images binaires.

Echéance

- Le laboratoire sera rendu au plus tard la veille du prochain laboratoire (**durée : 2 semaines**)

Validation du laboratoire

Selon la méthode énoncée en début de semestre, **chaque fin de session** doit aboutir à l'exécution d'un script qui transmettra le travail fourni. Chaque laboratoire s'accompagne d'un script de rendu appelé *rendu.sh* qu'il est **nécessaire** d'appeler à la fin de chaque session :

```
reds@reds2021:~/sye/sye21_lab02$ ./rendu.sh session
```

Cette commande va générer un dossier *rendu* qui contiendra les différences avec le dépôt de base.

Lorsque le laboratoire est terminé, il suffit **d'exécuter le script** en lui spécifiant **fin** comme paramètre :

```
reds@reds2021:~/sye/sye21_lab02$ ./rendu.sh fin
```

Cette commande va générer un dernier fichier de différences, puis archiver le dossier *rendu*. Il est demandé de **déposer cette archive** (*rendu.tar.gz*) **dans Moodle à la fin du laboratoire**.

Etape 1 - Mise à jour du dépôt (environnement)

La commande suivante permet de récupérer une mise à jour du dépôt pour la réalisation de ce laboratoire.

```
reds@reds2021:~/sye$ retrieve_lab sye21 lab02
```

Ceci va créer un dossier « **sye_lab02** » qui contiendra les fichiers nécessaires au laboratoire.

Etape 2 - Rajout d'un appel système dans SO3

Cette étape permettra l'introduction d'un nouvel appel système - appelé (directement) ***sys_fork2*** - permettant la création d'une copie de processus. Cet appel système sera une copie de l'appel système **sys_fork** avec un affichage d'informations supplémentaire. Ce qui nous intéresse surtout ici est le cheminement pour rajouter un nouvel appel système dans SO3.

a) En vous basant sur la théorie donnée lors du cours et des appels systèmes déjà implémentés dans SO3, rajoutez l'appel système sys_fork2. Le code de ce dernier est exactement le même que le code de l'appel système sys_fork. Il vous est demandé d'afficher un message provenant du kernel lors de l'appel à sys_fork2. Ce message aura la forme suivante :

```
fork2(): process test_fork2.elf with pid 2 forked to child with pid 3
```

⇒ L'équivalent de la fonction « **printf() »** dans le kernel est « **printk() »**.
⇒ La liste des fichiers à modifier est :

- so3/arch/arm32/include/asm/syscall.h
- so3/include/process.h
- so3/kernel/process.c
- so3/kernel/syscalls.c
- usr/lib/libc/include/syscall.h
- usr/lib/libc/crt0.S

À vous de trouver les modifications à faire et dans quel ordre.

⇒ Il est possible de tester le fonctionnement du syscall **sys_fork2** en exécutant le programme *test_fork2*.

b) Dans le fichier *rapport_lab02.md*, expliquez à quoi servent les différents ajouts et le cheminement complet pour ajouter un appel système.

Etape 3 – Jeu *Tic-Tac-Toe*

Cet exercice permet d'exercer l'exécution de plusieurs processus en parallèle. L'application qui servira de test est constituée des fichiers *tictactoe_gm.c* et *tictactoe_player.c* (dans *usr/src*). Il s'agit d'un squelette d'une implémentation du jeu *Tic-Tac-Toe* ou *Morpion* (plus d'informations ici : https://fr.wikipedia.org/wiki/Tic-tac-toe). Vous devez compléter ce squelette (voir indications *TO COMPLETE* dans le code).

L'application est composée de deux programmes : **tictactoe_gm** et **tictactoe_player**. Le premier représente l'IA (*Intelligence Artificielle*) ainsi que la gestion de la partie (affichage de la grille de jeu, détection de la victoire, etc…). Le second représente le joueur « humain ».

L'application qui doit être lancée depuis le *shell* est *tictactoe_gm*. Une fois en exécution, cette dernière devra exécuter l'application *tictactoe_player* en parallèle à l'aide de l'appel système **sys_fork2** implémenté dans l'étape précédente.

Indications

- L'algorithme de jeu de l'IA et une partie de sa gestion sont déjà implémentés.

- Une librairie s'occupe d'implémenter des bouts de code qui ne vous concernent pas. Cependant, il faudra que vous utilisiez les méthodes *ipc_player_argv1()* et *ipc_player_argv2()* à un moment donné. À vous de lire la documentation de ces fonctions et d'en faire bon usage.
- Les parties GM et Player communiquent à l'aide d'une librairie. La communication entre les deux entités est déjà implémentée.
- La saisie du mouvement du joueur doit être contrôlée. Les choix possibles sont les cases de 1 à 9.
- Des indications supplémentaires sur l'exécution sont données sur les pages suivantes.

Voici un exemple d'affichage :

```
so3% tictactoe_gm
Initial board:
   |   |
---+---+---
   |   |
---+---+---
   |   |
The valid moves are 1-9.

Start player's turn...
tictactoe% 9
End player's turn.
   |   |
---+---+---
   |   |
---+---+---
   |   | O

Start computer's turn...
   |   |
---+---+---
 X |   |
---+---+---
   |   | O
End computer's turn.

Start player's turn...
tictactoe% a
Invalid!
tictactoe% 10
Invalid!
tictactoe% 9
Invalid!
tictactoe% 3
   |   | O
---+---+---
 X |   |
---+---+---
   |   | O
```

```
End player's turn.

Start computer's turn...
```

```
   |   | O
---+---+---
 X | X |
---+---+---
   |   | O
End computer's turn.

Start player's turn...
tictactoe% 6
End player's turn.
   |   | O
---+---+---
 X | X | O
---+---+---
   |   | O
YOU WIN! New game (Y/N)?

...
```

Pseudo code du thread principal

```
Lancement des programmes IA et joueur
Boucle tant que le jeu n'est pas fini
        Tour joueur ?
        Oui :
                displayBeginPlayersTurn()
                Attend que le joueur a fini (attente active sur le status)
                displayEndPlayersTurn()
        Non :
                displayBeginComputersTurn()
                Attend que l'AI a fini (attente active sur le status)
                displayEndComputersTurn()

        Gagnant ?
        Oui :
                Afficher gagnant
                Stopper le jeu (Attendre la fin du processus fils et quitter)
Fin boucle
```

Etape 4 - Création et exécution de plusieurs processus en parallèle

Cet exercice permet d'exercer le démarrage et de gérer l'exécution de plusieurs processus. L'application qui servira de test est constitué du fichier « *usr/src/count_parallel.c* ».

a) Compléter le fichier *count_parallel.c* afin que celui-ci permette l'exécution de N enfants du programme *count.elf* (N passé en paramètre) en **parallèle**. Le programme *count.elf* s'attend à un paramètre *id* qui correspond à l'identifiant unique du processus enfant, $id \in [0; N[$.
b) Limiter le nombre de processus enfant à 15 au maximum.
c) Modifier le code afin que l'exécution de chaque *enfant* soit maintenant déterministe, c-à-d que chaque *enfant* s'exécute l'un après l'autre.

⇒ On demande à ce que le programme puisse être lancé avec les deux variantes à l'aide d'un argument au démarrage du programme, comme suit :

```
so3% count_parallel <N> p                # exécute l'application en version "parallèle"
so3% count_parallel <N> s                # exécute l'application en version "séquentielle"
```

⇒ Si l'exécution en parallèle des compteurs n'est pas visible, changer la constante *COUNT* dans le fichier *count.c* à 500 ou 1'000 jusqu'à voir les compteurs itérer simultanément.