IUT Orsay

DUT Informatique
Module Système S4-L
2010 / 2011

Travaux Pratiques n° 2 : Synchronisation

Nom(s) :
Groupe :
Date :

Objectifs : rappels sur la création de processus par la primitive `fork()` et synchronisation de processus par `wait()` et `exit()`. Maîtriser l'échange d'information et la synchronisation entre processus par tubes Unix.

1 Création de processus : primitive `fork()`

Nous utiliserons dans ce TP les primitives Unix suivantes étudiées en Système S2 :

```
#include <unistd.h>
pid_t fork(void);
pid_t getpid(void);
pid_t getppid(void);
```

1. La primitive `fork()` crée un processus, elle renvoie une valeur *n* qui indique :
   - si $n > 0$, on est dans le processus père, *n* vaut le numéro (le PID) du fils,
   - si $n = 0$, on est dans le processus fils,
   - si $n = -1$, fork a échoué, aucun nouveau processus n'a été créé.
2. La primitive `getpid()` retourne le numéro (le *pid*) du processus courant.
3. La primitive `getppid()` retourne le numéro (le *pid*) du processus père.

Les autres rappels nécessaires à la réalisation de ce TP sont fournis en notes de cours au début du TD n° 2. Par ailleurs il est vivement conseillé de se référer à votre cours de Système S2 ainsi qu'au polycopié *Primitives Système sous UNIX* distribué en première année. Bien sûr les pages de `man` pour `pipe` sont comme toujours de précieuses alliées.

2 Synchronisation par `wait()` et `exit()`

Il était une fois un programmeur débutant en programmation système sous UNIX qui avait une machine bi-core (avec deux processeurs). Ayant découvert la possibilité de créer plusieurs processus (et donc d'exploiter ses deux processeurs) il voulut s'en servir pour réaliser le travail suivant qu'il avait jusqu'à présent programmé de manière séquentielle : stocker dans une variable entière `v`, la valeur `f(x)+ g(x)` pour un `x` choisi, où `f` et `g` sont deux fonctions à argument entier renvoyant chacune `0` ou `1` mais dont les calculs, indépendants l'un de l'autre, sont très longs. Il écrivit donc le programme suivant en C (wait.c) :

```c
#include <stdio.h>
#include <sys/types.h>

int f(int x) { /* code de f ... */ }
int g(int x) { /* code de g ... */ }

int main() {
  int v=0, x;
  pid_t n;

  printf("Pid: %d\n", (int)getpid());

  /* Essai de calcul pour x=3 */
  x = 3;
  n = fork();
  if (n != 0)
    v = v + f(x);
  else
    v = v + g(x);

  printf("Valeur de v: %d, mon pid: %d\n", v, (int)getpid());
  return 0;
}
```

Ce programme est compilé sans erreur mais son exécution surprend le programmeur... Pour la suite, vous utiliserez les instructions suivantes comme corps des fonctions `f` et `g` :

```c
srand(getpid());
sleep(2);
return(random()%2);
```

Expliquez l'erreur du programmeur.

Donnez deux exemples d'exécutions possibles différentes.

Notre programmeur rencontre alors un collègue chevronné qui écrit une version correcte du programme. Pour résoudre le problème du retour de résultat du fils, il utilise le code de retour de la primitive `exit`, récupéré par `wait`. Un appel `exit(i)` termine en effet un processus et envoie au processus père l'entier `i` sur un octet : `(char)i` (inclure la bibliothèque `stdlib.h` pour avoir accès à la primitive `exit`). La primitive `wait` a le prototype suivant :

```
#include <sys/wait.h>
pid_t wait(int * etat);
```

À l'appel de `wait`, un processus père s'endort dans l'attente de la mort d'un de ses fils. À la mort d'un fils, `wait` retourne le pid du fils mort : on ne sait pas par avance qui va mourir mais on sait après coup qui est mort. Dans le cas où il n'y a plus (ou pas) de fils à attendre, `wait` retourne -1. En plus de son pid, le processus père récupère à l'adresse `etat` la valeur envoyée par `exit` et la cause de la mort du fils condensées en un seul entier (l'octet de poids fort contient la valeur renvoyée par le fils, et celui de poids faible contient la cause de la mort en cas de terminaison anormale). La macro `WEXITSTATUS(etat)` permet d'extraire la valeur d'`exit` seule (voir le fichier `/usr/include/sys/wait.h`).

Donnez l'algorithme d'une solution au problème en détaillant les actions du père et du fils.

Écrivez le programme C correspondant et **joignez le code source à votre compte rendu**.

Combien de temps dure une exécution de votre programme ?

Quelque temps plus tard, notre programmeur transmet son programme à un ami travaillant sur un autre ordinateur, vantant l'accroissement de performances de cette nouvelle version. Hélas, il est très déçu lorsque son ami lui indique que la nouvelle version est plus lente que la version séquentielle ! Dépité, notre héros retourne consulter son collègue expérimenté qui lui explique la raison de tout cela.

Proposez une explication.

3 Estimation de π

Le but de ce problème est de construire une application client-serveur utilisant les tubes comme moyen de communication pour trouver une approximation du nombre π.
La méthode utilisée pour calculer une valeur approchée de π, due au mathématicien allemand Gottfried Wilhelm Leibniz (1646-1716) dans les années 1670, appelée Série de Leibniz, est de réaliser une somme de nombreux termes pour calculer $\frac{\pi}{4}$:

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

Pour ce problème, un processus appelé *pi* va faire des appels à un autre processus appelé *leibniz* pour lui demander un nombre correspondant au $n^{ième}$ terme de la Série de Leibniz. Pour cela, *pi* enverra au travers d'un tube le nombre *n* à *leibniz*. Le processus *leibniz* calculera alors le nombre et l'enverra à son tour au travers d'un tube. Quand *pi* aura fait une demande et reçu sa réponse, il ajoutera le nombre reçu à sa somme. Il affichera après chaque demande de terme son évaluation de π qui devrait être de plus en plus précise.

On impose que *leibniz* soit le processus père et *pi* son processus fils. Décrivez l'organisation des communications entre les deux processus.

Dans ce schéma, qui est le client et qui est le serveur ?

Dans un premier temps on réalisera une application simplifiée où le processus fils *pi* enverra toutes les secondes à son processus père *leibniz* un nombre *n*. Le processus *leibniz* réagira en affichant le résultat du calcul du $n^{ième}$ terme. Rappels :
- Pour qu'un processus s'endorme une seconde, on fait appel à `sleep(1);`
- Le header nécessaire pour manipuler les tubes est `unistd.h`.
- Lorsqu'un processus n'utilise pas un des descripteurs d'un tube (lecture ou écriture), fermer ce descripteur est de la bonne programmation.

Réalisez cette première application, **vous joindrez le code source au compte-rendu**.

Complétez cette première application afin de réaliser le programme complet : créez un tube pour que *leibniz* puisse communiquer son résultat à *pi* et implémentez dans *pi* les calculs nécessaires à l'estimation du nombre π. Affichez le résultat aprés chaque estimation.

Réalisez cette seconde application, **vous joindrez le code source au compte-rendu**.

Lorsque le processus *pi* a fini d'envoyer des nombres au processus *leibniz*, si aucun test de retour de la primitive système `read` n'est fait pour arrêter le programme, le programme va tout de même s'arrêter quand il essaiera d'envoyer un résultat. Expliquez ce qui arrive.

Au bout de 100000 termes, combien de décimales de π sont correctes avec la Série de Leibniz ? Combien de temps cela prend-il à calculer (pour en avoir une idée relativement précise, ajoutez `time` devant la commande de lancement de votre programme) ?

4 Estimation plus précise de π (facultatif)

Lorsque l'on cherche à réaliser des application très performantes, en faisant appel par exemple à des serveurs très optimisés, il faut s'assurer que le temps gagné en temps de calcul n'est pas perdu en temps de communication. Ici on ne cherche pas à optimiser quoi que ce soit, mais on peut masquer le temps de communication en faisant effectuer au serveur des calculs plus longs. Une des m'ethodes les plus efficaces pour estimer π est une variation des Séries de Ramanujan, mises au point par le mathématicien indien Srinivasa Ramanujan (1887-1920) en 1910, que l'on doit aux frères mathématiciens américains d'origine ukrainième David et Gregory Chudnovsky. Cette série composée en 1987 est aujourd'hui utilisée dans le logiciel Mathematica et a permis plusieurs records de calcul de décimales de π :

$$\frac{1}{12\pi} = \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!(13591409 + 545140134n)}{(3n)!(n!)^3 640320^{3k+1}\sqrt{640320}}$$

Vous pourrez vérifier à quel point cette suite arrive très vite à une excellente précision du nombre π. Si vite que les 15 décimales prévues par le type `double` seront rapidement exactes et qu'il devriendra difficile (mais pas impossible pour un bon programmeur !) d'observer les décimales suivantes.
Les 15 premières décimales de π sont : $\pi = 3.14159\ 26535\ 89793$.

Au bout de combien de termes les 15 décimales sont exactes avec la Série de Chudnovsky ? Combien de temps cela prend-il à calculer ?

Commentaires personnels sur le TP (résultats attendus, difficultés, critiques etc.).