iUT ORSAY
Département Informatique

DUT Informatique
Module Système S4 C
2009 / 2010

Travaux Pratiques n° 2 : Communication

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 entre processus par tubes et signaux 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 (tubes et signaux)

Le but de ce problème est de construire une application client-serveur utilisant les signaux et les tubes comme moyens de communication pour trouver une approximation du nombre π.
La méthode utilisée pour calculer une valeur approchée de π, appelée méthode de Monte-Carlo, est d'effectuer des tirages aléatoires de nombres réels *x* et *y* tels qu'ils soient compris entre 0 et 1. Si *N* est le nombre de tirages de couples *x* et *y*, et *M* le nombre de tirages tels que $x^2 + y^2 \leq 1$, alors on a $\pi \simeq (4M)/N$.
Pour ce problème, un processus appelé *pi* va faire des appels à un autre processus appelé *tirage* pour lui demander un nombre aléatoire entre 0 et 1. Pour cela, *pi* enverra le signal `SIGUSR1` à *tirage*. Le processus *tirage* calculera alors le nombre et l'enverra au travers d'un tube. Quand *pi* aura fait une demande et reçu les réponses pour *x* et pour *y*, il calculera si $x^2 + y^2 \leq 1$. Il affichera après chaque demande de *x* et *y* son évaluation de π qui devrait être de plus en plus précise.

Est-ce que ce problème pouvait être résolu en utilisant uniquement les signaux comme moyen de communication (expliquez) ?

On impose que *tirage* 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 *tirage* le signal `SIGUSR1`. Le processus *tirage* réagira en affichant un nombre réel entre 0 et 1 tiré aléatoirement. Rappels :

- Pour qu'un processus s'endorme une seconde, on fait appel à `sleep(1);`
- L'instruction `aleatoire = ((float)random()/RAND_MAX);` place dans la variable de type `float` `aleatoire` un nombre réel entre 0 et 1 choisi aléatoirement.
- Les headers nécessaires pour manipuler les signaux sont `signal.h` et `unistd.h`.

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 *tirage* 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.
Pour rappel, lorsqu'un processus n'utilise pas ou plus un des descripteurs d'un tube (lecture ou écriture), fermer ce descripteur est de la bonne programmation.

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

4 Estimation de π (tubes seuls)

On cherche à écrire une nouvelle version du programme précédant sans utiliser de signaux mais uniquement des tubes.

Décrivez l'organisation des communications entre les deux processus.

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

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