M2 FEADéP – Systèmes et réseaux
TP2 Processus et tubes

Nicolas Chappe
École Normale Supérieure de Lyon

1 Un peu d'exclusion mutuelle

Pré-requis Threads et mutex de la bibliothèque pthreads

Exercice 1

On souhaite calculer une approximation de π à l'aide de la formule classique

$$\sum_{n\geq 1} \frac{1}{n^2} = \frac{\pi^2}{6}$$

1. Commencez par implanter un algorithme séquentiel à partir d'une boucle for faisant 100 millions d'itérations. À la fin, affichez l'approximation obtenue.
2. Parallélisez maintenant l'algorithme en créant 4 threads chacun chargé d'un quart du calcul. À la fin, affichez l'approximation obtenue depuis le thread principal et assurez-vous que le résultat est bien identique à celui du code séquentiel.
3. Essayez d'améliorer les performances de votre programme, à nombre de threads égal. Vous pourrez par exemple comparer les performances à l'aide de la commande `time`.

2 Processus

Fonctions utiles.

— `pid_t fork()` duplique le processus actuel dans un espace mémoire séparé (voir `man fork` pour plus de détails). La fonction retourne 0 dans le processus enfant et l'identifiant du processus (PID) enfant dans le processus parent.

— `pid_t waitpid(pid_t pid, int *wstatus, int options)` bloque le thread appelant jusqu'à ce que le processus enfant de PID donné termine son exécution.

Exercice 2

Cet exercice vise à écrire un programme qui permet de lancer des commandes en entrant leur nom ou leur chemin, à la manière d'un shell.

1. Documentez-vous sur la famille de fonctions `exec`, et écrivez un programme qui remplace le processus actuel par `ls`.
2. Écrivez un mini-shell, qui demande en boucle un nom de programme à l'utilisateur puis l'exécute. Vous aurez besoin des fonctions `fork`, `execlp` et `waitpid`.
3. Que se passe-t-il si l'on omet l'appel à `waitpid` ?
4. Ajoutez la possibilité de lancer une commande `c` en arrière-plan avec la syntaxe `c&`.

3 Tubes

Une fonctionnalité essentielle du shell est la notion de tube (ou *pipe* en anglais), qui s'utilise avec la syntaxe `commande1 | commande2` pour rediriger la sortie standard de `commande1` vers l'entrée standard de `commande2`. À l'origine de cette fonctionnalité, il y a la fonction C `pipe` qui peut s'utiliser de la manière suivante :

```
int pipefd[2];
pipe(pipefd);
FILE *fReception = fdopen(pipefd[0], "r");
FILE *fEnvoi = fdopen(pipefd[1], "w");

// Utilisation des fichiers...

fclose(fReception);
fclose(fEnvoi);
```

Ici, tout ce qui est écrit dans fEnvoi peut ensuite être lu dans fReception, formant ainsi un canal de communication unidirectionnel.

Exercice 3

Écrivez un programme dont le thread principal a accès à une variable `FILE *sortieMin`, qui recopie les chaînes de caractères qui lui sont données vers la sortie standard, mais en minuscules (utiliser `tolower`).

Exercice 4

Écrivez un programme qui lance `zenity --calendar` puis qui affiche sur la sortie standard quel jour, quel mois (en toutes lettres) et quelle année ont été sélectionnés. Essayez cette commande dans un terminal pour en comprendre le fonctionnement.

Exercice 5

Les objectifs de cet exercice sont de réimplanter la fonction C/POSIX `popen`, puis d'ajouter les tubes au mini-shell écrit dans la partie précédente.

1. Implantez une fonction `FILE *popenw(const char *command)` telle que après `f = popenw("c")`, tout ce qui est écrit dans f soit transmis à la commande c.
2. Implantez une fonction `FILE *popenr(const char *command)` telle que après `f = popenr("c")`, tout ce que la commande c écrit dans sa sortie standard soit transmis à f.
3. Implantez une fonction `void tube(const char *commande1, const char *commande2)` qui effectue l'équivalent de `commande1 | commande2` du shell.
4. Ajoutez à votre mini-shell la possibilité de lancer deux commandes `c1` et `c2` liées par un tube avec la syntaxe `c1|c2`.
5. Même question pour n tubes.

Vous aurez pour toutes ces questions besoin de l'appel système `dup2` pour rediriger l'entrée ou la sortie standard vers un tube.