U. Paris Diderot Programmation système avancée le 14 mai 2019 8h30-11h30

Documents autorisés :

(1) trois feuilles manuscrites format A4 recto/verso de notes personnelles, chaque page doit porter votre nom,

(2) les deux polys du cours.

Le sujet comporte 4 pages.

Votre code doit être écrit de façon **lisible**, *avec des indentations et des accolades appropriées permettant de voir la fin des blocs de code (fin de boucles, etc.). Vous pouvez, si vous le jugez utile, écrire des fonctions auxiliaires.*

Il est inutile d'écrire les includes. Une gestion d'erreurs, même minimaliste, sera appréciée (mais ce n'est pas le point le plus important).

Exercice 1 [signaux] Votre tâche consiste à modifier le programme suivant

```
1  void taskA(){
2    printf("task A\n");
3  }
4  void taskB(){
5    printf("task B\n");
6  }
7
8  int main(){
9
10   while( 1 ){
11     taskA();
12   }
13
14   taskB();
15   return 0;
16 }
```

de façon à ce que

(1) le signal `SIGUSR1` soit bloqué pendant l'exécution de `taskA()` (vous pouvez le bloquer avant la boucle `while`),

(2) après chaque exécution de `taskA()` vérifiez si le signal `SIGUSR1` est pendant et si c'est le cas il faut sortir de la boucle `while` pour exécuter `taskB()`,

(3) faire en sorte que l'exécution de `taskB()` ne puisse pas être interrompue par aucun signal [1]

Exercice 2 [processus, signaux] On considère le programme suivant :

```
1  int main(int argc, char **argv){
2
3    avant();
4
5    pid_t pid = fork();
```

1. à l'exception, bien sûr, des signaux qui ne peuvent jamais être ni bloqués ni ignorés.

```
 6     if( pid == -1 )
 7        exit( 1 );
 8     if( pid == 0 ){
 9        execlp("tache","tache", argv[1], (char *)NULL);
10     }
11
12     apres();
13
14  }
```

Le programme `tache` contient une boucle infinie donc une fois lancé il est sensé de ne jamais s'arrêter.

La fonction `apres()` contient aussi une boucle infinie donc quand le processus père fait appel à `apres()` le contrôle reste dans la fonction `apres()`.

Modifiez le programme ci-dessus pour assurer le comportement suivant :

(1) si le processus fils n'arrive pas à faire exécuter `tache` (exec échoue) alors le fils doit faire en sorte que le père termine (le père n'a rien à faire sans le fils) et le fils lui-même doit terminer,

(2) Hélas le programme `tache` est buggé et le processus fils peut disparaître mystérieusement (enfin, pas complètement, il devient zombie). Dans ce cas le processus père doit
   — supprimer le fils zombie,
   — relancer un nouveau fils qui exécutera `tache`,
   — ensuite le père doit revenir à son activité.

Remarque : Modification de code de la fonction `apres()` n'est pas une option valable (et de toute façon on ne voit pas ce code).

Indication : Quel signal est envoyé vers le processus père quand le fils termine ? Ce signal est par défaut ignoré par le père.

Exercice 3 [fichiers] On suppose que le banque stocke les soldes de tous les comptes dans un fichier. Ce fichier contient des nombres de type `long`, écrit en binaire, et stockés un après l'autre. On suppose que i-ème nombre dans le fichier correspond au solde du compte numéro i (les comptes sont numérotés à partir de 0 et le solde est toujours en euros sans centimes).

Ecrire la fonction

```
long transfert(const char *fcomptes, unsigned int deb, unsigned int cred, unsigned
                                                                    int s)
```

qui transfert la somme de `s` euros du compte `deb` vers le compte `cred`. Le paramètre `fcpmptes` est le nom de fichier de comptes.

Si l'opération réussit la fonction `transfert` retourne `s` sinon elle retourne -1.

Le transfert échoue (et les deux comptes ne changent pas de solde) si une des conditions suivantes est satisfaite :

(1) le solde de `deb` deviendrait négatif à la suite du transfert,

(2) un des comptes n'existe pas (par exemple le fichier contient 100 comptes mais on essaie de créditer le compte numéro 200),

Vous devez implémenter l'algorithme qui lit le solde de deux comptes **et seulement le solde de deux compte** `deb` et `cred`, effectue l'opération et sauvegarde les deux soldes dans le fichier.

L'implémentation qui lit les solde de tous les comptes dans un grand tableau et après les modifications sauvegarde tout le tableau de soldes n'est ni acceptable ni accepté.

Exercice 4 [verrous de fichier] Dans cet exercice on supposera que la fonction `transfert` de l'exercice précédent peut être utilisée et appelée en parallèle par plusieurs processus. Pour que le fichier de comptes reste toujours consistent il faudra poser les verrous avant la lecture de soldes et lever les verrous après la modification de soldes.

Ajoutez dans la fonction `transfert` les opération appropriées sur les verrous.

Remarque 1. Pas la peine de recopier la totalité de la fonction de l'exercice précédent, il suffit juste écrire le code qui pose et enlève les verrous et indiquer clairement où vous l'ajouterez dans la fonction de l'exercice précédent.

Remarque 2. Pour obtenir plus de parallélisme on préfère les verrous posés sur les deux soldes modifiés par `transfer()`. Une solution qui verrouille tout le fichier apportera moins de points.

Question. On suppose qu'on pose les verrous juste sur les comptes concernés par le transfert. Est-ce que plusieurs processus effectuant le transfert en même temps peuvent provoquer un interblocage? Si oui donner un exemple simple.

Avez-vous une idée (simple) comment éviter interblocage?

Exercice 5 [question exec] Un programme contient le fragment de code suivant :

```
1 execl("/bin/echo", "echo", "bonjour", (char *) 0);
2 char *a="au-revoir\n";
3 write(1, a, strlen( a ) );
```

Si le contrôle de programme arrive à la ligne contenant `execl` alors quel(s) texte(s) sera (seront) écrit(s) sur la sortie standard? Justifiez en une seul phrase. (Tout le monde sait ce que fait la commande `echo`, n'est-ce pas?)

Exercice 6 [tubes] Le processus père et fils communiquent par les tubes anonymes.

Le père lit les caractères sur l'entrée standard et les envoie dans un tube vers le fils. Le fils applique à chaque caractère la fonction `char transformer(char c)` et envoie le caractère obtenu vers le père en utilisant aussi un tube (est-ce que cela peut-être le même tube?).

Le père affiche sur la sortie standard les caractères qu'il reçoit de son fils.

Implementer ce comportement.

Vous devez donner une implémentation complète, avec la création de tubes, de processus fils etc.

Et n'oubliez pas de fermer les descripteurs non-utilisés aussi bien chez le père que chez le fils.

Exercice 7 [redirections] On suppose que nous avons à notre disposition un programme `transformer` qui lit les caractères un par un sur l'entrée standard et après une transformation écrit chaque caractère transformé sur la sortie standard. Ce programme est donné, il est compilé et on assume que le fichier exécutable `transformer` se trouve dans le répertoire courant. Nous n'avons pas d'accès au code source de `transformer`, on sait juste que pour chaque caractère lu sur le descripteur 0 le programme `transformer` écrit un caractère sur le descripteur 1.

Le but de l'exercice est d'écrire un autre programme (écrire sa fonction `main`) dans lequel le processus père lit des caractères dans un fichier dont le nom est l'unique argument de la commande (élément du vecteur `argv` de `main`).

Le processus père envoie chaque caractère lu vers le processus fils en utilisant un tube anonyme. Le processus fils exécute le programme `transformer` pour transformer chaque caractère envoyé par le père.

L'exercice ressemble un peu à l'exercice précédent mais cette fois, pour simplifier, on ne demande plus que le processus fils renvoie les caractères transformés vers le processus père, le fils écrit les caractères sur la sortie standard. Par contre cette fois le fils n'exécute pas une fonction mais il exécute un programme.

Comme dans l'exercice précédent, n'oubliez de fermer les descripteurs non-utilisés.

Le programme que vous devez écrire doit implémenter le comportement décrit ci-dessus (la création de fils qui exécutera le programme `transformer` et la communication par le tube entre le processus père et le fils).

Exercice 8 [projection mémoire] Implémenter la commande (écrire la fonction `main`)

mysimplegrep s n file

qui cherche la première occurrence de la chaîne de caractère `s` dans le fichier `file`. La recherche doit commencer à partir de `n`-ième caractère du fichier `file` (les caractère avant la position `n` ne sont pas pris en compte). Comme d'habitude les positions de caractères dans le fichier sont comptées à partir de 0.

La commande affiche sur la sortie standard la position de `s` dans le fichier où -1 si aucune occurrence n'est retrouvée.

Exemple. Supposons que le fichier `f` contient le texte :

```
alaalaala
```

avec 3 occurrences de mot `ala` qui commencent sur les positions 0,3, 6. Dans ce cas

- `mysimplegrep ala 0 f` écrira 0. L'argument 0 de la commande indique qu'on commence la recherche depuis le début du fichier.
- `mysimplegrep ala 1 f` écrira 3. Le paramètre 1 indique que la recherche commence à la position 1, juste après le premier caractère de `f`.
- Et finalement `mysimplegrep ala 4 f` écrira 6, tandis que `mysimplegrep ala 7 f` écrira -1.

Votre implémentation doit projeter le fichier (ou, encore mieux, la partie utile de fichier) dans la mémoire et faire la recherche en utilisant la projection.
