Université de Strasbourg
Licence 2 Informatique

TP noté

Programmation système
2021/2022 – printemps

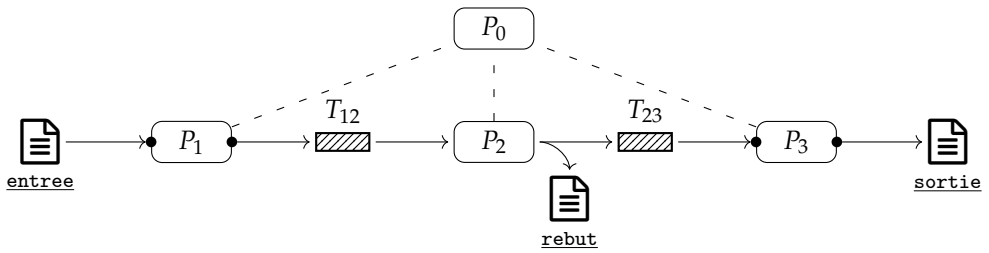
Conditions : 1h30, travail individuel ; seuls documents autorisés : Moodle et travaux pratiques personnels ; utilisez l'archive fournie, et rendez un unique fichier nommé `fltr.c` dans le délai.

Vous devez écrire un programme appelé `fltr`, que l'on exécute avec 3 arguments :

```
./fltr entree rebut sortie
```

où `entree` désigne un fichier existant, et `rebut` et `sortie` des fichiers produits au cours du traitement.

Le traitement effectué par `fltr` est réalisé à l'aide de trois processus fils P_1, P_2 et P_3, et de deux tubes T_{12} et T_{23}. Les données circulent le long des flèches. Le schéma du système doit être le suivant :



Description des processus impliqués :

— le processus P_0, qui exécute le programme `fltr` :
  – crée tous les objets nécessaires, puis attend que ses processus fils se terminent
  – se termine avec succès si et seulement si les trois processus fils se terminent avec succès (voir ci-dessous)
— le processus P_1 :
  – exécute le programme standard `"tr"` avec les arguments `".#"` et `"01"`
  – redirige son entrée standard vers le fichier `entree`
  – redirige sa sortie standard vers le coté en écriture du tube T_{12}
— le processus P_2 :
  – lit des données à partir du coté en lecture du tube T_{12}
  – écrit des données soit dans le coté en écriture du tube T_{23}, soit dans le fichier `rebut`
  – le format des données et le critère de décision sont décrits en détail ci-dessous
— le processus P_3 :
  – exécute le programme standard `"cat"` avec l'argument `"-n"`
  – redirige son entrée standard vers le coté en lecture du tube T_{23}
  – redirige sa sortie standard vers le fichier `sortie`

Détails du processus P_2 : Le processus P_2 n'exécute pas un programme externe, il faut donc écrire son code. Voici les détails du traitement qu'il effectue :

— les données lues à partir du tube T_{12} sont des blocs de 8 octets exactement, dont les valeurs peuvent être quelconques : par exemple | 30 | 31 | 30 | 31 | 31 | 30 | 31 | 0A | (les octets 0 et 6 sont grisés ici – voir point suivant)
— dans un bloc lu, si la valeur de l'octet 0 (le premier) est égale à la valeur de l'octet 6 (l'avant-dernier), alors le bloc entier est écrit dans le fichier `rebut` ; dans le cas contraire, le bloc est écrit dans le tube T_{23}
— toute lecture fournissant un bloc incomplet (c'est-à-dire de taille 1 à 7 octets) est considérée comme une erreur, et termine immédiatement, avec échec, l'exécution du processus P_2

Le processus P_2 lit des blocs de données tant qu'il y en a, puis s'arrête avec succès.

Code de retour du programme : Le processus principal P_0 doit avoir pour code de retour :

0 : si et seulement si les trois processus fils se terminent avec succès
2 : si l'un au moins des trois processus fils est arrêté par la réception d'un signal
1 : dans tous les autres cas (aucun processus arrêté par un signal, mais l'un au moins échoue)

Consignes :

— Vous devez compléter le programme `fltr.c` fourni : c'est ce fichier que vous devez rendre sur Moodle
— Vous devez utiliser uniquement des primitives système
— Vous devez utiliser le `makefile` fourni pour compiler et tester votre programme :

make	pour compiler
make test-small	pour un test simple (10 blocs en entrée) ; n'allez pas plus loin tant que ce test échoue
make test-large	pour un test plus long (qui dure environ 10 secondes)
make test-errs	pour tester la robustesse de votre programme

A Gestion des erreurs

Vous devez évidemment tester tous les appels système. Le fichier `fltr.c` contient une fonction et une macro pour vous y aider. Nous vous recommandons de les utiliser :

— la fonction `raler()` prend en argument un chaîne de caractères, affiche un message débutant par cette chaîne, puis arrête le programme sur un échec

— la macro `CHECK()` permet de tester si un appel système renvoie `-1` : on peut par exemple écrire simplement

```
CHECK (stat (&st));
```

si l'appel renvoie `-1`, `raler()` est appelée (ce n'est qu'un exemple, vous n'avez évidemment pas besoin de `stat` dans ce TP) ; cette macro ne teste que `-1`, elle ne convient pas pour `read` ou `write`

Vous n'êtes pas obligés d'utiliser `raler` et `CHECK`, mais vous devez évidemment tester tous les appels système.

B Tests

Le *makefile* permet de lancer les tests. On lance le premier test avec

```
make test-small
```

ou bien directement, avec

```
./test.sh small
```

Les deux formes sont parfaitement identiques. Vous pouvez aussi effectuer ce test « à la main » avec

```
./fltr small.entree small.rebut small.sortie
cmp small.rebut small.rebut.ref
cmp small.sortie small.sortie.ref
```

Les fichiers `small.entree`, `small.rebut.ref` et `small.sortie.ref` sont fournis dans l'archive.

Ce premier test vérifie que votre programme fonctionne correctement. Si votre programme se bloque, le script affiche un message après 5 secondes. Si ce test échoue, il n'est pas recommandé de lancer les autres tests.

Le test `large` lance votre programme avec un fichier d'entrée contenant plusieurs millions de blocs. Ce test peut prendre entre 10 et 20 secondes. Votre programme est interrompu automatiquement après 30 secondes.

Le test `errs` teste un certain nombre de cas problématiques : format d'entrée incorrect, erreurs dans les appels système etc. Il vous permet de vous assurer que votre programme se comporte correctement dans (presque) toutes les situations. Si ce test échoue, vous avez oublié de vérifier quelque chose dans votre programme.

C Format des blocs de données utilisées dans les tests

Attention : vous n'avez pas besoin de comprendre en détail la totalité du traitement (en particulier, ce que font *exactement* P_1 et P_3) pour écrire le programme `fltr`.

Dans la table ci-dessous, seule la colonne centrale vous concerne : elle décrit le format des données lues et écrites par P_2. La colonne de gauche décrit le format des données lues par P_1 ; celle de droite le format des données écrites par P_3.

`entree`	T_{12} et T_{23}, et `rebut`*	`sortie`
`. . . # # # # \n`	`0 0 0 1 1 1 1 \n`	`␣ ␣ ␣ ␣ ␣ 1 \t 0 0 0 1 1 1 1 \n`
`. # . . # . . \n`	`0 1 0 0 1 0 0 \n`*	`␣ ␣ ␣ ␣ ␣ 2 \t 1 0 0 1 1 1 0 \n`
`# . . # # # . \n`	`1 0 0 1 1 1 0 \n`	
`# # # . # # # \n`	`1 1 1 0 1 1 1 \n`*	

La commande « `tr .# 01` » lit son entrée standard caractère par caractère, transforme le caractère `'.'` en `'0'`, le caractère `'#'` en `'1'`, conserve intacts tous les autres caractères, et écrit le résultat sur sa sortie standard.

La commande « `cat -n` » lit son entrée standard ligne par ligne, préfixe chaque ligne par son numéro (à partir de 1) écrit sur 6 caractères et suivi d'une tabulation, et écrit le résultat sur sa sortie standard.