Système – ENS L3 Info.
Année 2015-2016

Examen
2 juin 2016

L'énoncé est composé de 4 pages. Cette épreuve est prévue pour une durée de 3h. Les notes de cours et de TDs imprimées ou sur votre ordinateur sont autorisées.

Le code pourra être écrit en OCaml ou en C. La lisibilité du code et la qualité de la rédaction seront prises en compte.

1 `rm -rf` : suppression d'une hiérarchie des fichiers

Question 1. Écrivez une fonction pour supprimer un chemin que ce soit un fichier ou un répertoire. Cette fonction doit prendre comme argument le chemin à supprimer :

```
val rm : string -> unit
```

Utilisez les fonctions `Unix.stat`, `Unix.unlink` et `Unix.rmdir`. Vous pouvez assumer qu'un répertoire ne contient que des fichiers réguliers (`S_REG`) et d'autres répertoires (`S_DIR`).

2 `moinsync` : synchronisation des fichiers

Il s'agit de développer des parties d'une commande `moinsync` qui va dupliquer un répertoire d'un client sur un serveur. Cette synchronisation n'est faite que dans un sens : il n'y a pas de résolution de conflits entre fichiers. Nous allons suivre le schéma du programme `rsync` d'Unix, mais en réalisant une version simplifiée.

Un utilisateur lance le programme en mode *client* en donnant deux arguments :

```
moinsync  /tmp/source  di.ens.fr:12345//var/dest
```

Dans cet exemple, le client connecte au serveur `di.ens.fr` sur le port `12345` et commence une synchronisation du répertoire local `/tmp/source` avec le répertoire cible au serveur `/var/dest`. On suppose que le programme a déjà été lancé en mode *serveur* sur la machine `di.ens.fr`[1], qu'il écoute sur le port `12345` et que le répertoire `/var/dest` existe.

Le protocole a deux phases principales. Dans la première, le client envoie la liste de tous ses fichiers au serveur. Dans la deuxième, le serveur et le client travaillent ensemble pour transférer les contenus des fichiers.

Dans les questions suivantes, vous pouvez ignorer les permissions, les propriétaires et les groupes, ainsi que les questions de sécurité. Pour les erreurs critiques, il suffira de lever une exception. Vous pouvez supposer que les systèmes de fichier ne contiennent que des fichiers réguliers (`S_REG`) et des répertoires (`S_DIR`). On fera l'hypothèse également que les appels système ne sont pas interrompus.

1. Normalement, la commande `rsync` lance le serveur en utilisant le protocole ssh.

2.1 Phase 1 : transfert des chemins relatifs

Question 2. Implémenter la fonction du client

```
val send_file_list : string -> out_channel -> unit
```

qui prend le chemin source, par ex., `/tmp/source`, et un canal (du module OCaml `Pervasives`) qui va au serveur et sur lequel il faut envoyer les chemins relatifs des répertoires et des fichiers qui se trouvent sous le chemin source.

La fonction doit envoyer des lignes de texte au serveur. Chaque ligne contient un chemin relatif. Les chemins relatifs des répertoires doivent être terminés par un '`/`'. Une ligne contenant le chemin d'un fichier doit être suivie par deux lignes : la première contient la date de modification du fichier (la partie entière de `st_mtime`) et la deuxième contient la taille du fichier. La liste est terminée par une ligne vide.

Question 3. Implémenter la fonction du serveur

```
val recv_file_list : string -> in_channel -> (string * int) list
```

qui prend le chemin cible, par ex., `/var/dest`, et un canal qui vient du client et qui reçoit les chemins relatifs envoyés par le client.

Cette fonction doit lire les données envoyées par `send_file_list`, créer des répertoires si nécessaire et rendre une liste des chemins à mettre à jour avec leurs dates de modification. Un fichier est à mettre à jour soit parce qu'il n'existe pas sur le serveur, soit parce que sa date de modification ou sa taille diffèrent entre client et serveur.

Après avoir reçu la liste des chemins du client, le serveur doit supprimer les fichiers et les répertoires qui existent de son côté mais qui n'apparaissent pas dans la liste. Vous n'avez pas à implémenter cette fonctionnalité.

2.2 Phase 2 : transfert des contenus des fichiers

Dans la deuxième phase, le serveur et le client communiquent pour transférer les blocs de contenu du client au serveur en essayant de minimiser le trafic sur le réseau. Admettons que la taille d'un bloc soit définie par la constante suivante.

```
let block_size = 1024
```

Pour les questions suivantes, vous supposerez que les fonctions `read` donnent toujours le nombre d'octets demandés sauf à la fin d'un fichier ou donne tous les octets qui restent (et par la suite, rends `0` comme résultat).

Admettons qu'il existe une fonction

```
val checksum : string -> int -> int -> string
```

qui prend un buffer (`string` ou `bytes`), un décalage et une longueur et qui rend une chaîne de 32 caractères représentant la somme de contrôle correspondant (comme le module `Digest` d'OCaml).

Question 4. Implémenter la fonction du serveur

```
val generator : string -> (string * int) list -> out_channel -> unit
```

qui prend le chemin source, la liste des fichiers calculée par `recv_file_list` et un canal qui va au client. Pour chaque fichier, elle doit envoyer le chemin relatif sur une ligne, suivi par une ligne pour chaque bloc du fichier contenant sa somme de contrôle, et finalement une ligne vide. Si le fichier n'existe pas sur le serveur, la fonction n'envoie que le chemin et la ligne vide. Une ligne vide supplémentaire doit être envoyée après la liste.

Question 5. Implémenter la fonction du client

```
val sender : string -> in_channel -> out_channel -> unit
```

qui prend le chemin source comme premier argument et qui reçoit sur le canal d'entrée les données envoyées du serveur par la fonction `generator`. Chaque chemin relatif reçu doit être répété sur le canal de sortie. Ensuite la fonction doit calculer les sommes de contrôle du fichier local et les comparer aux celles reçues du serveur. Pour chaque bloc où les sommes de contrôle ne sont pas égales, la fonction envoie une ligne avec le numéro du bloc [2], une ligne avec la taille du bloc t et puis le contenu du bloc (t octets). Si nécessaire, la fonction envoie un bloc de taille 0 pour la première bloc qui existe sur le serveur mais qui n'existe pas dans le fichier local. Après avoir envoyé tous les blocs qui diffèrent, il faut envoyer une ligne vide.

Question 6. Implémenter la fonction du serveur

```
val receiver : string -> (string * int) list -> in_channel -> unit
```

qui prend le chemin cible, la liste des fichiers calculée par `recv_file_list` et un canal d'entrée en argument. Cette fonction reçoit les données envoyées par `sender` sur le canal d'entrée. Elle doit reconstituer un fichier local à partir des blocs reçus sur le canal et des blocs du fichier local (s'il y en a un).

2.3 Le programme final

Question 7. Combiner les réponses aux questions précédentes pour implémenter la fonction client suivante.

```
val client : string -> string -> in_channel -> out_channel -> unit
```

Elle prend en entrée un chemin source, un chemin cible, un canal qui va au serveur et un canal qui vient du serveur. Cette fonction implémente le côté client du protocole.

Question 8. Combiner les réponses aux questions précédentes pour implémenter la fonction suivante.

```
val server : in_channel -> out_channel -> unit
```

qui prend en entrée un canal qui vient du client et un canal qui va au client et qui implémente le côté serveur du protocole. Pour profiter du pipelining et pour éviter un interblocage, les fonctions `generator` et `receiver` doivent s'exécuter en parallèle.

Question 9. En fait, le protocole décrit ci-dessus n'est pas très efficace en pratique. Pourquoi ? Comment corriger ce défaut sans trop s'éloigner des idées principales du protocole ?

2. Son décalage dans le fichier local divisé par `block_size`.

3 Le bar à margarita

Il s'agit d'un problème de partage des ressources. Il y a trois ressources : la *tequila* (en verre de 3,5 cl), le *cointreau* (en verre de 2,0 cl), et le *jus de citron* (en verre de 1,5 cl). Il faut un verre de chacun pour faire une margarita. Il y a aussi trois *buveurs* de margarita et un *barman*. Chaque buveur a une quantité illimitée d'une ressource que les deux autres n'ont pas. Ils ne partagent jamais entre eux. À chaque tour, le barman choisit deux des ressources et les met sur le comptoir. Il attend qu'elles soient toutes les deux prises avant de recommencer. Les buveurs essaient de prendre les deux ressources qui leur manquent pour en faire une margarita. Il ne faut pas que la fête s'arrête (pas d'interblocage) !

Chaque ressource est modélisée par un sémaphore. Pour rappel, un sémaphore est initialisé avec un état s et permet deux opérations :

`wait` : décrémente l'état ($s' = s - 1$), l'appelant est bloqué si $s' < 0$.
`post` : incrémente l'état ($s' = s + 1$), débloque un processus qui attend s'il y en a un et si $s' \geq 0$.

Les ressources sur le comptoir sont modélisées par trois sémaphores :

```
tequila = semaphore(0)
cointreau = semaphore(0)
jusdecitron = semaphore(0)
```

Il y a aussi un sémaphore pour demander au barman de mettre deux ressources sur le comptoir.

```
barmanSem = semaphore(1)
```

Le barman est modélisé par les trois processus suivants.

```
let barman1 () =
  while true do
    barmanSem.wait();
    tequila.post();
    cointreau.post()
  done
```

```
let barman2 () =
  while true do
    barmanSem.wait();
    cointreau.post();
    jusdecitron.post()
  done
```

```
let barman3 () =
  while true do
    barmanSem.wait();
    tequila.post();
    jusdecitron.post()
  done
```

Question 10. En utilisant les processus séquentiels, les sémaphores et les variables globales, définir le système complet. Vous n'avez pas le droit de changer le modèle du barman.